pyRSD Documentation

Release 0.1.17

Nick Hand

Feb 20, 2018

Getting Started

Accurate predictions for the clustering of galaxies in redshift-space in Python

pyRSD is a Python package for computing the theoretical predictions of the redshift-space power spectrum of galaxies. The package also includes functionality for performing Bayesian parameter estimation using the MCMC sampling technique or Maximum a posteriori estimation using the LBFGS algorithm to perform the nonlinear optimization.

Note: The theoretical models used in this paper are described in more detail in Hand et al. 2017. Please cite this work if you use this package in your research.

CHAPTER 1

Index

Getting Started

1.1 Install pyRSD

pyRSD can be installed using the `conda` utility or by installing from source.

1.1.1 Conda

The easiest installation method uses the `conda` utility, as part of the Anaconda package manager. We have pre-built binaries available that are compatible with Linux and macOS platforms. The package can be installed via:

```
conda install -c nickhand -c astropy pyrsd
```

The package is available for Python versions 2.7, 3.5, and 3.6.

1.1.2 Install From Source

pyRSD can also be installed directly from source:

```
# clone the github source
git clone https://github.com/nickhand/pyRSD.git
cd pyRSD

# run the install
pip install .
```

1.1.3 Test

Test whether or not the installation succeeded by importing the module in IPython:

```
import pyRSD
```

1.2 Overview

pyRSD provides functionality for computing complex, theoretical models of the galaxy power spectrum in redshift space. The package has two main modules:

1. `pyRSD.rsd`

   The module responsible for evaluating theoretical power spectra and related quantities for an input cosmology specified by the user.

2. `pyRSD.rsdfit`

   The module responsible for performing parameter estimation; it uses the theory models in `pyRSD.rsd` and finds the best-fit parameters describing an input data set.

1.2.1 The pyRSD.rsd module

This module provides the ability to compute several theoretical power spectrum quantities. These include

1. *`pyRSD.rsd.GalaxySpectrum`*

   The galaxy power spectrum in redshift space. See *this section* for more details.

2. `pyRSD.rsd.QuasarSpectrum`

   The quasar power spectrum in redshift space. See *this section* for more details

3. `pyRSD.rsd.hzpt`

   A module for computing dark matter power spectra using Halo Zel'dovich Perturbation Theory. See *this section* for more details.

4. `pyRSD.pygcl`

   A module for interfacing with the CLASS Boltzmann code and computing various clustering quantities using the CLASS transfer function. This is a swig-wrapped C++ module that computes most of the perturbation theory and other numerically intensive calculations on which the pyRSD models are based. See *this section* for more details.

1.2.2 The pyRSD.rsdfit module

This module handles parameter estimation, fitting the theoretical models provided in the `pyRSD.rsd` module to data provided by the user. There are two ways parameter estimation can be performed:

1. Monte Carlo Markov Chain (MCMC)

   The full posterior distribution of the model parameters can be found using the `emcee` Python MCMC package.

2. Nonlinear optimization via the LBFGS algorithm

The best-fit parameters can be found by maximizing the likelihood distribution, which is performed using the well-known LBFGS algorithm.

1.3 Quickstart

The pyRSD package provides a number of examples in order to get users up and running quickly. Once these examples have been downloaded, the user can start running their own parameter fits using the example data and parameter files. The executable responsible for downloading the pyRSD examples is `pyrsd-quickstart`, which has the following calling sequence:

```
$ pyrsd-quickstart -h
usage: pyrsd-quickstart [-h]
                        {galaxy/survey-poles,galaxy/periodic-poles,galaxy/periodic-
↪pkmu}
                        dirname

download example data and parameter files to get up and running with pyRSD

positional arguments:
  {galaxy/survey-poles,galaxy/periodic-poles,galaxy/periodic-pkmu}
                        which example to download
  dirname               the output directory to save the downloaded files; if
                        it doesn't exist it will be created

optional arguments:
  -h, --help            show this help message and exit
```

The first argument provided to the command is the name of the example to download, which should be of the following:

Example Name	Description
galaxy/periodic-pkmu	Fitting $P(k, \mu)$ galaxy data from a periodic box simulation
galaxy/periodic-poles	Fitting $P_\ell(k)$ galaxy data from a periodic box simulation
galaxy/survey-poles	Fitting $P_\ell(k)$ galaxy data from a simulation with a realistic survey window function

And the second argument is the name of the directory to download the files too. So, for example, to download the files from the `periodic-poles` examples to a directory called `pyRSD-example` simply do

```
pyrsd-quickstart galaxy/periodic-poles pyRSD-example
```

1.3.1 Running Parameter Fits

With the example downloaded, the user can run MCMC or nonlinear optimization fits using the default model parametrization with the `rsdfit` command. See *The Free Parameters* for a table of the default free parameters and *The Constrained Parameters* for a table of the default constrained parameters.

For example, to run 10 nonlinear optimization steps (using the LBFGS optimization algorithm), you can do

```
rsdfit nlopt -m pyRSD-example/model.npy -p pyRSD-example/params.dat -o pyRSD-example-
↪results -i 10
```

or to run 10 MCMC iterations (using 30 `emcee` walkers), you can do

```
rsdfit mcmc -m pyRSD-example/model.npy -p pyRSD-example/params.dat -o pyRSD-example-
↪results -i 10 -w 30
```

The number of iterations has been set to 10 here just for illustration purporse. Typically, the LBFGS algoritm will take $\sim 100 - 200$ steps to converge, and the mcmc algorithm will need 1000s of iterations to fully explore the posterior distributions of the parameters.

1.3.2 Analyzing the Fit Results

The `rsdfit` saves the best-fit parameter set to a numpy `.npz` file in the directory specified via the `-o` output, which is `pyRSD-example-results` in the example above. There are two Python objects in pyRSD that can read these files, depending on the type of fit that was run. For `mcmc` fits, use the *pyRSD.rsdfit.results.EmceeResults* class and for `nlopt` fits, use the *pyRSD.rsdfit.results.LBFGSResults* class.

For example, to explore the fitting results from a `mcmc` fit

```
In [1]: from pyRSD.rsdfit.results import EmceeResults, LBFGSResults

In [2]: mcmc_results = EmceeResults.from_npz('mcmc_result.npz')

# print out a summary of the parameters, with mean values and 68% and 95% intervals
In [3]: print(mcmc_results)
Free parameters [ median (+/-68%) (+/-95%) ]
_________________
<Parameter Nsat_mult:      2.423 (+0.2082 -0.1809) (+0.4361 -0.3432)>
<Parameter alpha_par:      1.009 (+0.00682 -0.007681) (+0.01353 -0.01379)>
<Parameter alpha_perp:     1.005 (+0.004241 -0.004049) (+0.008281 -0.008406)>
<Parameter b1_cA:          1.999 (+0.05749 -0.05801) (+0.1126 -0.1265)>
<Parameter f:              0.867 (+0.02897 -0.02502) (+0.05642 -0.05073)>
<Parameter f1h_sBsB:       3.569 (+0.5521 -0.6841) (+1.224 -1.396)>
<Parameter fs:             0.1434 (+0.007553 -0.008046) (+0.01554 -0.01544)>
<Parameter fsB:            0.4662 (+0.08146 -0.0792) (+0.1685 -0.1606)>
<Parameter gamma_b1sA:     1.31 (+0.1001 -0.1032) (+0.2119 -0.2205)>
<Parameter gamma_b1sB:     2.343 (+0.1661 -0.181) (+0.3225 -0.3636)>
<Parameter sigma8_z:       0.5411 (+0.01224 -0.01362) (+0.02611 -0.02452)>
<Parameter sigma_c:        0.9297 (+0.06205 -0.06473) (+0.1126 -0.1177)>
<Parameter sigma_sA:       3.443 (+0.2779 -0.2702) (+0.5436 -0.5378)>

Constrained parameters [ median (+/-68%) (+/-95%) ]
_____________________
<Parameter NsBsB:          5.178e+04 (+1.721e+04 -1.36e+04) (+4.16e+04 -2.44e+04)>
<Parameter b1_sA:          2.617 (+0.169 -0.1761) (+0.3037 -0.3725)>
<Parameter sigma_sB:       5.71 (+0.3067 -0.2659) (+0.6624 -0.4928)>
<Parameter NcBs:           2.261e+04 (+2900 -2096) (+6130 -4001)>
<Parameter b1_sB:          4.686 (+0.2679 -0.3235) (+0.5596 -0.672)>
<Parameter b1_cB:          3.175 (+0.1504 -0.1736) (+0.2942 -0.3522)>
<Parameter fcB:            0.122 (+0.01336 -0.0149) (+0.02856 -0.02761)>
<Parameter b1_c:           2.141 (+0.04721 -0.04684) (+0.08834 -0.08337)>
<Parameter b1:             2.344 (+0.05369 -0.04764) (+0.1051 -0.1053)>
<Parameter fsigma8:        0.4689 (+0.009673 -0.009076) (+0.01866 -0.01848)>
<Parameter b1_s:           3.575 (+0.1907 -0.2037) (+0.3854 -0.4186)>
<Parameter alpha:          1.006 (+0.004107 -0.004331) (+0.007706 -0.0083)>
<Parameter epsilon:        0.00125 (+0.002298 -0.002462) (+0.004696 -0.004652)>
<Parameter b1sigma8:       1.268 (+0.007575 -0.008006) (+0.01531 -0.01495)>
<Parameter F_AP:           1.004 (+0.006927 -0.007386) (+0.01419 -0.01393)>
```

(continues on next page)

(continued from previous page)

```
# access parameters like a dictionary
In [4]: fsat = mcmc_results['fs']

In [5]: print(fsat.median)
0.14339263725236592
```

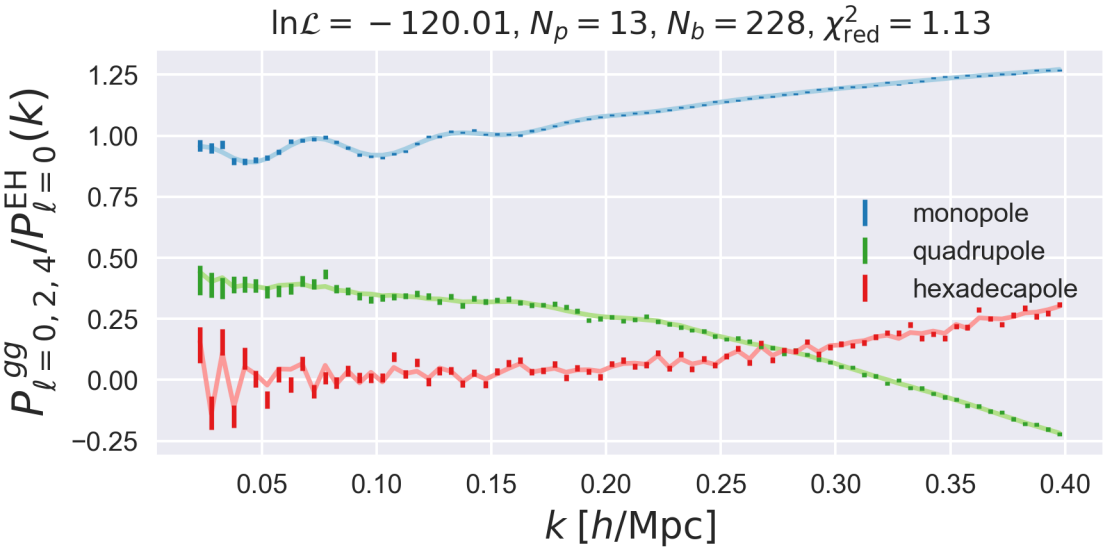
and to explore the fitting results from a `nlopt` fit

```
In [6]: nlopt_results = LBFGSResults.from_npz('nlopt_result.npz')

# print out a summary of the parameters, with best-fit values
In [7]: print(nlopt_results)
minimum chi2 = 120.57745038002743

Free parameters [ mean ]
________________
Nsat_mult       : 2.4324089012014207
alpha_par       : 1.0072991239946898
alpha_perp      : 1.0048333693698863
b1_cA           : 2.0068377074748778
f               : 0.8735346371321935
f1h_sBsB        : 3.6140366462767153
fs              : 0.14175570928456935
fsB             : 0.4782779433107577
gamma_b1sA      : 1.31213984320964
gamma_b1sB      : 2.3651886897525896
sigma8_z        : 0.5363993024676117
sigma_c         : 0.9194602113178405
sigma_sA        : 3.420304679802984

Constrained parameters [ mean ]
_____________________________
NsBsB           : 51736.1
b1_sA           : 2.6332517
NcBs            : 23183.68
fcB             : 0.11864934
sigma_sB        : 5.739127
b1_sB           : 4.74655
b1_cB           : 3.2117057
epsilon         : 0.0008172965
b1_c            : 2.1497946
b1sigma8        : 1.2667638
alpha           : 1.0056546
F_AP            : 1.0024539
b1_s            : 3.6439955
fsigma8         : 0.46856338
b1              : 2.3616061

# access best-fit values like a dictionary
In [8]: fsat = nlopt_results['fs']

In [9]: print(fsat)
0.14175570928456935
```

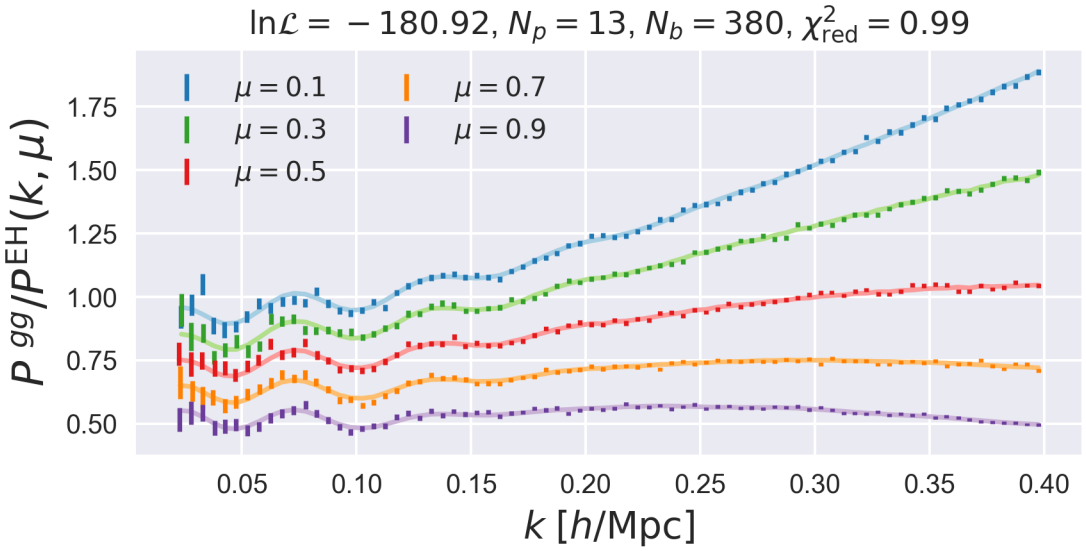
1.3.3 Comparing the Best-fit Model to Data

Users can compare the best-fitting model to the data by loading the results of a fitting run using the *pyRSD.rsdfit.FittingDriver*. We can easily initialize this object by passing the directory where the results were written to the *pyRSD.rsdfit.FittingDriver.from_directory* function. For the example data downloaded above, we can explore both the data and theory simulataneously using the included result file `nlopt_result.npz`:

```
from pyRSD.rsdfit import FittingDriver

# load the model and results into one object
d = FittingDriver.from_directory('pyRSD-example', model_file='pyRSD-example/model.npy
↪', results_file='pyRSD-example/nlopt_result.npz')

# set the fit results
d.set_fit_results()

# the best-fit log probability (likelihood + priors)
print(d.lnprob())

# the best-fit chi2
print(d.chi2())

# the best-fit reduced chi2
print(d.reduced_chi2())

# make a plot of the data vs the theory
d.plot()
show()
```



In this plot, we show the monopole, quadrupole, and hexadecapole normalized by the smooth, no-wiggle Eisenstein and Hu monopole. All of the above steps are identical if we are analyzing $P(k, \mu)$ data rather than $P_\ell(k)$ data. For example, if the `periodic-pkmu` example is downloaded, running the function `FittingDriver.plot()` using the included result file `nlopt_result.npz` produces the following figure:



This plot shows the best-fit theory and data for 5 wide μ bins, normalized by the linear Kaiser $P(k, \mu)$, using the no-wiggle Eisenstein and Hu linear power spectrum.

RSD

The **RSD** module deals with computing the theoretical power spectrum predictions, given an input cosmology specified by the user.

- *Specifying the Cosmology*
- *Interfacing with CLASS*
- *Halo Zel'dovich Perturbation Theory*
- *Galaxy Power Spectrum*
- *Quasar Power Spectrum*

1.4 Specifying the Cosmology

The `pyRSD.rsd.cosmology` module allows users to specify the desired cosmological parameters, and it also provides the functionality for users to interface with the CLASS CMB Boltzmann code.

1.4.1 Overview

Users can specify cosmological parameters by creating a new `Cosmology` object or by using one of the builtin cosmologies (see *Available Cosmologies*).

When constructing a new Cosmology object, parameter values should be specified as keyword parameters. The parameters that can be specified are:

Parameter	Description
H0	The Hubble constant at z=0, in km/s/Mpc
Om0	The matter density/critical density at z=0
Ob0	The baryon density/critical density at z=0
Ode0	The dark energy density/critical density at z=0
w0	The dark energy equation of state
Tcmb0	The temperature of the CMB in K at z=0
Neff	The the effective number of neutrino species
m_nu	The mass of neutrino species in eV
sigma8	The the mass variance on the scale of R=8 Mpc/h at z=0, which sets the normalization of the linear power spectrum
n_s	The the spectral index of the primoridal power spectrum
flat	if True, automatically set Ode0 such that Ok0 is zero

Note: The *pyRSD.rsd.cosmology.Cosmology* class is nearly identical to the `astropy.cosmology.FLRW` object, with the addition of the `n_s` and `sigma8` attributes

Examples

```
In [1]: from pyRSD.rsd import cosmology

# initialize a new Cosmology
In [2]: cosmo = cosmology.Cosmology(H0=70, sigma8=0.80, n_s=0.96)

# access parameters as attribute or key entry
In [3]: print(cosmo['sigma8'], cosmo.sigma8)
0.8 0.8

# compute the comoving distance to z = 0.4
In [4]: Dz = cosmo.comoving_distance(0.4)

In [5]: print(Dz)
1547.1248846885328 Mpc
```

The Cosmology class is read-only; changes to the parameters should be performed with the *clone()* function, which creates a copy of the class, with any specified changes.

```
In [6]: new_cosmo = cosmo.clone(sigma8=0.85, Om0=0.27)

# compare sigma8
In [7]: print(cosmo.sigma8, new_cosmo.sigma8)
0.8 0.85

# compare Om0
In [8]: print(cosmo.Om0, new_cosmo.Om0)
\\\\\\\\0.31 0.27

# everything else stays the same
In [9]: print(cosmo.n_s, new_cosmo.n_s)
\\\\\\\\\\\\\\\\\\0.96 0.96
```

1.4.2 API

Top level user functions:

Cosmology([H0, Om0, Ob0, Ode0, w0, Tcmb0, ...])	Dict-like object for cosmological parameters and related calculations
Cosmology.clone(**kwargs)	Returns a copy of this object, potentially with some changes.
Cosmology.from_astropy(cosmo[, n_s, sigma8])	Return a *Cosmology* instance from an astropy cosmology
Cosmology.to_class([transfer, linear_power_file])	Convert the object to a *pyRSD.pygcl.Cosmology* instance in

There are builtin, default Cosmology objects available:

Name	Source	H0	Om	Flat
WMAP5	Komatsu et al. 2009	70.2	0.277	Yes
WMAP7	Komatsu et al. 2011	70.4	0.272	Yes
WMAP9	Hinshaw et al. 2013	69.3	0.287	Yes
Planck13	Planck Collab 2013, Paper XVI	67.8	0.307	Yes
Planck15	Planck Collab 2015, Paper XIII	67.7	0.307	Yes

The Cosmology class inherits the following **attributes** from the `astropy.cosmology.FLRW` class:

H0	Return the Hubble constant as an *~astropy.units.Quantity* at z=0
Neff	Number of effective neutrino species
Ob0	Omega baryon; baryonic matter density/critical density at z=0
Ode0	Omega dark energy; dark energy density/critical density at z=0
Odm0	Omega dark matter; dark matter density/critical density at z=0
Ogamma0	Omega gamma; the density/critical density of photons at z=0
Ok0	Omega curvature; the effective curvature density/critical density
Om0	Omega matter; matter density/critical density at z=0
Onu0	Omega nu; the density/critical density of neutrinos at z=0
Tcmb0	Temperature of the CMB as *~astropy.units.Quantity* at z=0
Tnu0	Temperature of the neutrino background as *~astropy.units.Quantity* at z=0
critical_density0	Critical density as *~astropy.units.Quantity* at z=0
h	Dimensionless Hubble constant: h = H_0 / 100 [km/sec/Mpc]
has_massive_nu	Does this cosmology have at least one massive neutrino species?
hubble_distance	Hubble distance as *~astropy.units.Quantity*
hubble_time	Hubble time as *~astropy.units.Quantity*
m_nu	Mass of neutrino species

The Cosmology class inherits the following **methods** from the `astropy.cosmology.FLRW` class:

H(z)	Hubble parameter (km/s/Mpc) at redshift z.
Ob(z)	Return the density parameter for baryonic matter at redshift z.
Ode(z)	Return the density parameter for dark energy at redshift z.
Odm(z)	Return the density parameter for dark matter at redshift z.
Ogamma(z)	Return the density parameter for photons at redshift z.
Ok(z)	Return the equivalent density parameter for curvature at redshift z.
Om(z)	Return the density parameter for non-relativistic matter at redshift z.
Onu(z)	Return the density parameter for neutrinos at redshift z.
Tcmb(z)	Return the CMB temperature at redshift z.
Tnu(z)	Return the neutrino temperature at redshift z.
abs_distance_integrand(z)	Integrand of the absorption distance.
absorption_distance(z)	Absorption distance at redshift z.
age(z)	Age of the universe in Gyr at redshift z.
angular_diameter_distance(z)	Angular diameter distance in Mpc at a given redshift.
angular_diameter_distance_z1z2(z1, z2)	Angular diameter distance between objects at 2 redshifts.
arcsec_per_kpc_comoving(z)	Angular separation in arcsec corresponding to a comoving kpc at redshift z.
arcsec_per_kpc_proper(z)	Angular separation in arcsec corresponding to a proper kpc at redshift z.
comoving_distance(z)	Comoving line-of-sight distance in Mpc at a given redshift.
comoving_transverse_distance(z)	Comoving transverse distance in Mpc at a given redshift.
comoving_volume(z)	Comoving volume in cubic Mpc at redshift z.
critical_density(z)	Critical density in grams per cubic cm at redshift z.
de_density_scale(z)	Evaluates the redshift dependence of the dark energy density.
differential_comoving_volume(z)	Differential comoving volume at redshift z.
distmod(z)	Distance modulus at redshift z.
efunc(z)	Function used to calculate H(z), the Hubble parameter.
inv_efunc(z)	Inverse of efunc.
kpc_comoving_per_arcmin(z)	Separation in transverse comoving kpc corresponding to an arcminute at redshift z.
kpc_proper_per_arcmin(z)	Separation in transverse proper kpc corresponding to an arcminute at redshift z.
lookback_distance(z)	The lookback distance is the light travel time distance to a given redshift.
lookback_time(z)	Lookback time in Gyr to redshift z.
lookback_time_integrand(z)	Integrand of the lookback time.
luminosity_distance(z)	Luminosity distance in Mpc at redshift z.
nu_relative_density(z)	Neutrino density function relative to the energy density in photons.
scale_factor(z)	Scale factor at redshift z.
w(z)	The dark energy equation of state.

Available Cosmologies

```
cosmology.Planck13 = {'H0':  67.77, 'Neff':  3.046, 'Ob0':  0.048252, 'Om0':  0.30712, 'Tcm
cosmology.Planck15 = {'H0':  67.74, 'Neff':  3.046, 'Ob0':  0.0486, 'Om0':  0.3075, 'Tcmb0
```

```
cosmology.WMAP5 = {'H0':  70.2, 'Neff':  3.04, 'Ob0':  0.0459, 'Om0':  0.277, 'Tcmb0':  2.
cosmology.WMAP7 = {'H0':  70.4, 'Neff':  3.04, 'Ob0':  0.0455, 'Om0':  0.272, 'Tcmb0':  2.
cosmology.WMAP9 = {'H0':  69.32, 'Neff':  3.04, 'Ob0':  0.04628, 'Om0':  0.2865, 'Tcmb0':
```

pyRSD.rsd.cosmology.Cosmology

class pyRSD.rsd.cosmology.**Cosmology**(*H0=67.6*, *Om0=0.31*, *Ob0=0.0486*, *Ode0=0.69*, *w0=-1.0*, *Tcmb0=2.7255*, *Neff=3.04*, *m_nu=0.0*, *n_s=0.9667*, *sigma8=0.8159*, *flat=False*, *name=None*)

Dict-like object for cosmological parameters and related calculations

An extension of the `astropy.cosmology` framework that can store additional, orthogonal parameters and behaves like a read-only dictionary

The class relies on `astropy.cosmology` as the underlying "engine" for calculation of cosmological quantities. This "engine" is stored as `engine` and supports `LambdaCDM` and `wCDM`, and their flat equivalents

Any attributes or functions of the underlying astropy engine can be directly accessed as attributes or keys of this class

---

Note: A default set of units is assumed, so attributes stored internally as `astropy.units.Quantity` instances will be returned here as numpy arrays. Those units are:

- temperature: `K`
- distance: `Mpc`
- density: `g/cm^3`
- neutrino mass: `eV`
- time: `Gyr`
- H0: `Mpc/km/s`

Warning: This class does not currently support a non-constant dark energy equation of state

clone(***kwargs*)

Returns a copy of this object, potentially with some changes.

Returns **newcos** : Subclass of FLRW

A new instance of this class with the specified changes.

Notes

This assumes that the values of all constructor arguments are available as properties, which is true of all the provided subclasses but may not be true of user-provided ones. You can't change the type of class, so this can't be used to change between flat and non-flat. If no modifications are requested, then a reference to this object is returned.

Examples

To make a copy of the Planck15 cosmology with a different Omega_m and a new name:

```
>>> from astropy.cosmology import Planck15
>>> cosmo = Cosmology.from_astropy(Planck15)
>>> newcos = cosmo.clone(name="Modified Planck 2013", Om0=0.35)
```

classmethod from_astropy(*cosmo*, *n_s=0.9667*, *sigma8=0.8159*, ***kwargs*)

Return a *Cosmology* instance from an astropy cosmology

Parameters **cosmo** : subclass of `astropy.cosmology.FLRW`

the astropy cosmology instance

****kwargs :**

extra key/value parameters to store in the dictionary

to_class(*transfer=0*, *linear_power_file=None*, ***class_config*)

Convert the object to a *pyRSD.pygcl.Cosmology* instance in order to interface with the CLASS code

Parameters ****class_config** : key/value pairs

keywords to pass to the CLASS engine; defaults are *z_max_pk=2.0* and *P_k_max_h/Mpc=20.0*

Returns **cosmo** : pygcl.Cosmology

the pygcl Cosmology object which interfaces with CLASS

1.5 Interfacing with CLASS

The main power spectrum calculations performed in pyRSD require the linear matter transfer function. We provide an interface to the CLASS CMB Boltzmann code via the `pyRSD.pygcl` module.

The `pyRSD.pygcl` module is the **General Cosmology Library**, a swig-generated Python wrapper of a C++ library that provides useful cosmological functionality, including an interface to the CLASS code.

1.5.1 Overview

The CLASS Cosmology object

Although the main pyRSD modules require the use of the main *pyRSD.rsd.cosmology.Cosmology* class, the `pyRSD.pygcl` module relies on the *pyRSD.pygcl.Cosmology* class. A *pyRSD.pygcl.Cosmology* object can be easily initialized from a *pyRSD.rsd.cosmology.Cosmology* object via the *to_class()* function, as

```
In [1]: from pyRSD.rsd.cosmology import Planck15

In [2]: from pyRSD import pygcl

In [3]: class_cosmo = Planck15.to_class()

In [4]: print(class_cosmo)
<pyRSD.pygcl.Cosmology; proxy of <Swig Object of type 'Cosmology *' at 0x7fde08cf2480>
↪ >
```

Computing Background Quantities

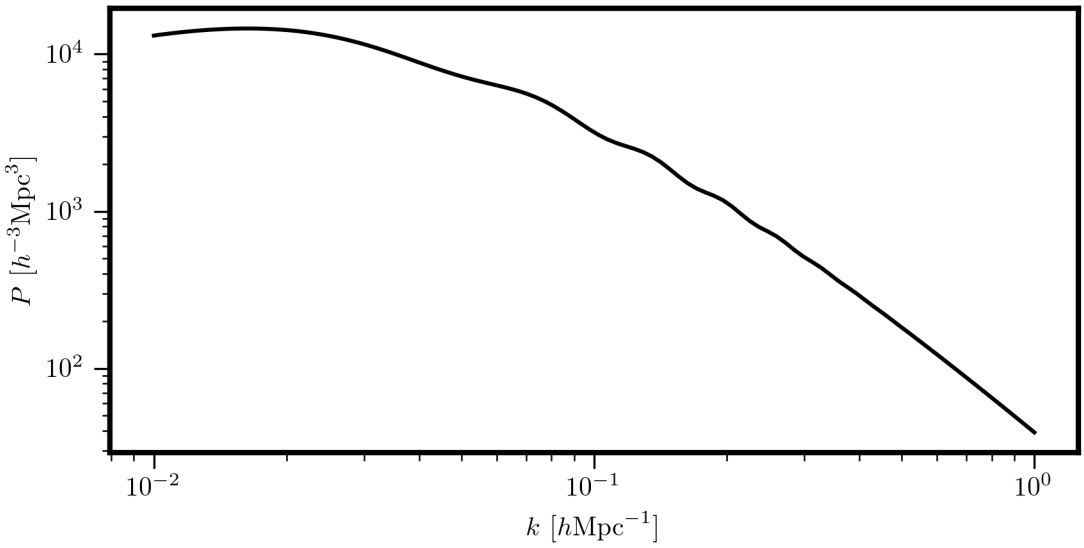
The *pyRSD.pygcl.Cosmology* calls the CLASS code to compute various cosmological parameters and background quantities as a function of redshift See *the API* for the full list of available quantities that CLASS can compute. For example, to compute the growth rate as a function of redshift

```
In [5]: z = numpy.linspace(0, 3, 100)

In [6]: growth = class_cosmo.f_z(z)
```

The Linear Power Spectrum

Most importantly for pyRSD, the `pyRSD.pygcl` module includes functionality to compute the linear matter power spectrum using CLASS. The main object for this calculation is the *pyRSD.pygcl.LinearPS* class, which can be initialized as

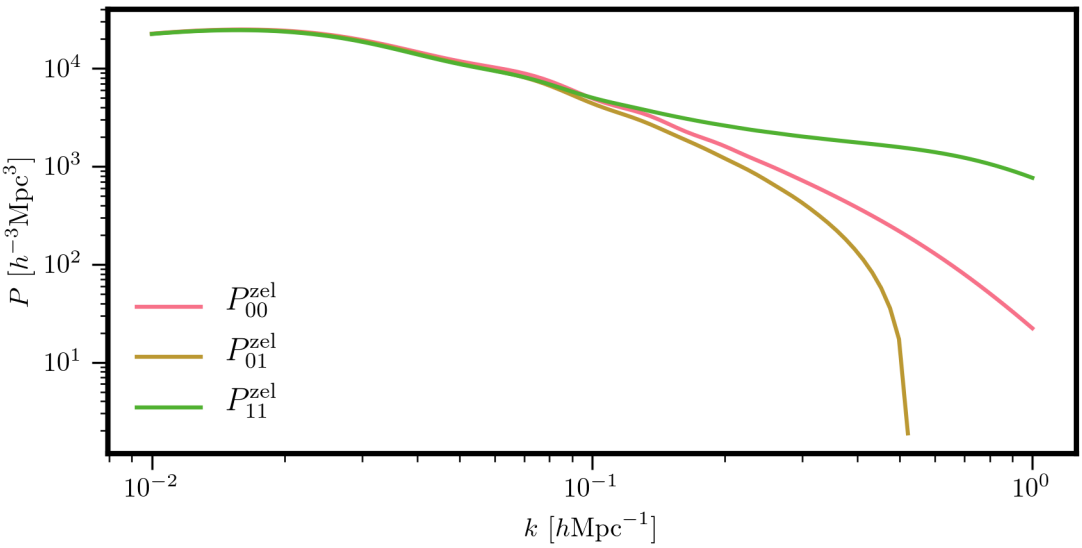
```
# initialize at z = 0
Plin = pygcl.LinearPS(class_cosmo, 0)

# renormalize to different SetSigma8AtZ
Plin.SetSigma8AtZ(0.62)

# evaluate at k
k = numpy.logspace(-2, 0, 100)
Pk = Plin(k)

# plot
plt.loglog(k, Pk, c='k')
```



Zel'dovich Power Spectra

The `pyRSD.pygcl` module can also be used to directly compute power spectra in the Zel'dovich approximation. For example,

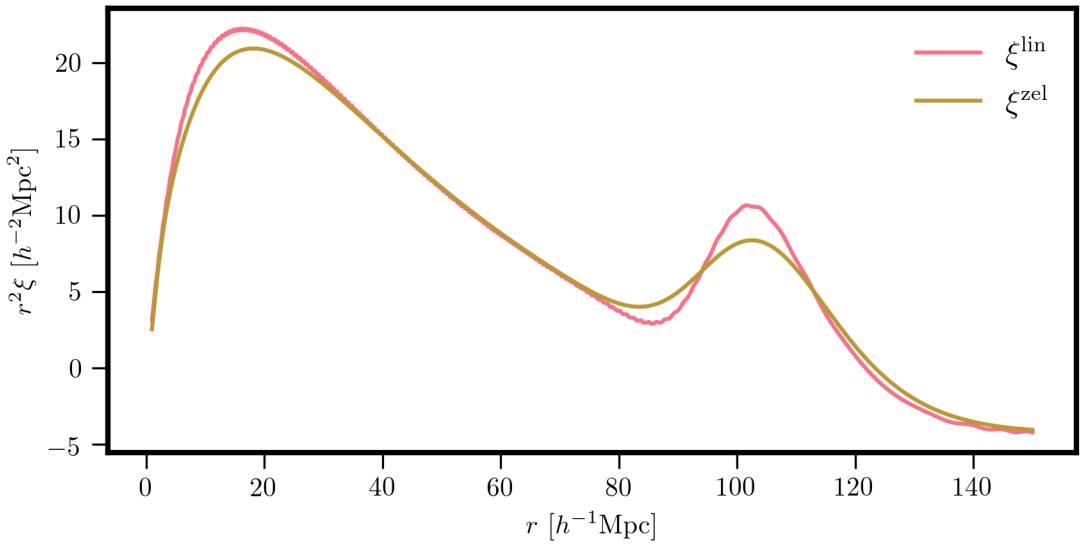
```
# density auto power
P00 = pygcl.ZeldovichP00(class_cosmo, 0)

# density - radial momentum cross power
P01 = pygcl.ZeldovichP01(class_cosmo, 0)

# radial momentum auto power
P11 = pygcl.ZeldovichP11(class_cosmo, 0)

# plot
k = numpy.logspace(-2, 0, 100)
plt.loglog(k, P00(k), label=r'$P_{00}^\mathrm{zel}$')
plt.loglog(k, P01(k), label=r'$P_{01}^\mathrm{zel}$')
plt.loglog(k, P11(k), label=r'$P_{11}^\mathrm{zel}$')
```



The Correlation Function

The `pyRSD.pygcl` module also includes functionality for computing the linear and Zel'dovich correlation functions. This is computed by taking the Fourier transform of the power spectrum using FFTLog.

```
# linear correlation function
CF = pygcl.CorrelationFunction(Plin)

# Zeldovich CF at z = 0.55
CF_zel = pygcl.ZeldovichCF(class_cosmo, 0.55)

# plot
r = numpy.logspace(0, numpy.log10(150), 1000)
plt.plot(r, r**2 * CF(r), label=r'$\xi^\mathrm{lin}$')
plt.plot(r, r**2 * CF_zel(r), label=r'$\xi^\mathrm{zel}$')
```



1.5.2 API

The Cosmology object

Methods that return various cosmological parameters:

class pyRSD.pygcl.**Cosmology**(**args*)
Proxy of C++ Cosmology class.

H0()
the present-day Hubble constant in units of km/s/Mpc

h()
the dimensionless Hubble constant

Tcmb()
CMB temperature today in Kelvin

Omega0_b()
present-day baryon density parameter

Omega0_cdm()
present-day cold dark matter density fraction

Omega0_ur()
present-day ultra-relativistic neutrino density fraction

Omega0_m()
present-day non-relativistic density fraction

Omega0_r()
present-day relativistic density fraction

Omega0_g()
present-day photon density fraction

Omega0_lambda()
present-day cosmological constant density fraction

Omega0_fld()
present-day dark energy fluid density fraction (valid if Omega0_lambda is unspecified)

Omega0_k()
present-day curvature density fraction

w0_fld()
present-day fluid equation of state parameter

wa_fld()
present-day equation of state derivative

n_s()
the spectral index of the primordial power spectrum

k_pivot()
the pivot scale in 1/Mpc

A_s()
scalar amplitude = curvature power spectrum at pivot scale

ln_1e10_A_s()
convenience function returns log (1e10*A_s)

sigma8()
convenience function to return sigma8 at z = 0

k_max()
maximum k value computed in h/Mpc

k_min()
minimum k value computed in h/Mpc

z_drag()
the baryon drag redshift

rs_drag()
the comoving sound horizon at the baryon drag redshifts

tau_reio()
the reionization optical depth

z_reio()
the redshift of reionization

rho_crit(*cgs=False*)
the critical density at z = 0 in units of h^2 M_sun / Mpc^3 if cgs = False, or in units of h^2 g / cm^3

EvaluateTransfer(*Cosmology self*, *double k*) → double

Methods that return background quantities as a function of redshift:

pyRSD.pygcl.**f_z**(*z*)
the logarithmic growth rate, dlnD/dlna, at z

pyRSD.pygcl.**H_z**(*z*)
the Hubble parameter at z in km/s/Mpc

pyRSD.pygcl.**Da_z**(*z*)
the angular diameter distance to z in Mpc – this is Dm/(1+z)

pyRSD.pygcl.**Dc_z**(*z*)
the conformal distance to z in the flat case in Mpc

pyRSD.pygcl.**Dm_z**(*z*)
the comoving radius coordinate in Mpc, which is equal to the conformal distance in the flat case

pyRSD.pygcl.**D_z**(*z*)
the growth function D(z) / D(0) (normalized to unity at z = 0)

pyRSD.pygcl.**Sigma8_z**(*z*)
the scalar amplitude at z, equal to sigma8 * D(z)

pyRSD.pygcl.**Omega_m_z**(*z*)
Omega0_m as a function of z

pyRSD.pygcl.**rho_bar_z**(*z*, *cgs=False*)
the mean matter density in units of h^2 M_sun / Mpc^3 if cgs = False, or in units of g / cm^3

pyRSD.pygcl.**rho_crit_z**(*z*, *cgs=False*)
the critical matter density in units of h^2 M_sun / Mpc^3 if cgs = False, or in units of g / cm^3

pyRSD.pygcl.**dV**(*z*)
the comoving volume element per unit solid angle per unit redshift in Gpc^3

pyRSD.pygcl.**V**(*zmin*, *zmax*, *Nz=1024*)
the comoving volume between two redshifts (full sky)

Power spectrum objects

class pyRSD.pygcl.**LinearPS**(**args*)
Proxy of C++ LinearPS class.

Compute the linear power spectrum, using CLASS

__init__(*pygcl.Cosmology cosmo*, *float z=0*)
initialize the linear power spectrum for a given cosmology and redshift

__call__(*k*)
evaluate the linear power spectrum at the wavenumber `k`, where `k` is in units of h/Mpc

SetSigma8AtZ(*sigma8_z*)
set the normalization of the power spectrum via setting $\sigma_8(z)$

class pyRSD.pygcl.**ZeldovichP00**(**args*)
Proxy of C++ ZeldovichP00 class.

Compute the density auto power spectrum in the Zel'dovich approximation

__init__(*pygcl.Cosmology cosmo*, *float z*, *bool approx_lowk=False*)
initialize the class for a given cosmology and redshift; if `approx_lowk` is True, use a low `k` approximation of the Zel'dovich approximation

__call__(*k*)
evaluate the Zel'dovich power spectrum at the wavenumber `k`, where `k` is in units of h/Mpc

SetSigma8AtZ(*sigma8_z*)
set the normalization of the power spectrum via setting $\sigma_8(z)$

class pyRSD.pygcl.**ZeldovichP01**(**args*)
Proxy of C++ ZeldovichP01 class.

Compute the density - radial momentum cross power spectrum in the Zel'dovich approximation

__init__(*pygcl.Cosmology cosmo*, *float z*, *bool approx_lowk=False*)
initialize the class for a given cosmology and redshift; if `approx_lowk` is True, use a low `k` approximation of the Zel'dovich approximation

__call__(*k*)
evaluate the Zel'dovich power spectrum at the wavenumber `k`, where `k` is in units of h/Mpc

SetSigma8AtZ(*sigma8_z*)
set the normalization of the power spectrum via setting $\sigma_8(z)$

class pyRSD.pygcl.**ZeldovichP11**(**args*)
Proxy of C++ ZeldovichP11 class.

Compute the radial momentum auto power spectrum in the Zel'dovich approximation

__init__(*pygcl.Cosmology cosmo*, *float z*, *bool approx_lowk=False*)
initialize the class for a given cosmology and redshift; if `approx_lowk` is True, use a low `k` approximation of the Zel'dovich approximation

__call__(*k*)
evaluate the Zel'dovich power spectrum at the wavenumber `k`, where `k` is in units of h/Mpc

SetSigma8AtZ(*sigma8_z*)
set the normalization of the power spectrum via setting $\sigma_8(z)$

Correlation function objects

class pyRSD.pygcl.**CorrelationFunction**(**args*)
Proxy of C++ CorrelationFunction class.

Compute the linear correlation function by Fourier transforming the linear power spectrum

__init__(*pygcl.LinearPS plin*, *kmin=1e-4*, *kmax=10*)
initialize the class from a linear power spectrum object; `kmin` and `kmax` correspond to the limits of the numerical integration when doing the Fourier transform.

__call__(*r*)
evaluate the correlation function at the separation `r`, where `r` is in units of Mpc/h

class pyRSD.pygcl.**ZeldovichCF**(**args*)
Proxy of C++ ZeldovichCF class.

Compute the density auto correlation function in the Zel'dovich approximation

__init__(*pygcl.Cosmology cosmo*, *float z*, *kmin=1e-4*, *kmax=10*)
initialize the class for a given cosmology and redshift; `kmin` and `kmax` correspond to the limits of the numerical integration when doing the Fourier transform.

__call__(*r*)
evaluate the Zel'dovich correlation function at the separation `r`, where `r` is in units of Mpc/h

SetSigma8AtZ(*sigma8_z*)
set the normalization of the correlation function via setting $\sigma_8(z)$

1.6 Halo Zel'dovich Perturbation Theory

The `pyRSD.rsd.hzpt` module allows users to compute various clustering quantities using Halo Zel'dovich Perturbation Theory (HZPT)

1.6.1 Overview

The `pyRSD.rsd.hzpt` module provides functionality for computing various power spectrum and correlation function quantities using Halo Zel'dovich Perturbation Theory (HZPT). See Seljak and Vlah 2015 for an introduction to HZPT.

The Fourier space quantities can be computed from the following classes:

Name	Quantity
HaloZeldovichP00	The dark matter auto power spectrum
HaloZeldovichP01	The dark matter density - radial momentum cross correlation
HaloZeldovichP11	The mu^4 contribution to the radial momentum auto spectrum
HaloZeldovichPhm	The halo - dark matter cross power

And the configuration space quantities can be computed from the following classes:

Name	Quantity
HaloZeldovichCF00	The dark matter auto correlation function
HaloZeldovichCFhm	The halo - dark matter cross correlation function

Initialization

An HZPT instance can be initialized by specifying a cosmology via a *pyRSD.rsd.cosmology.Cosmology* object and a redshift `z`.

Note: The objects should be initialized at a specific redshift by passing the `z` parameter. Once created, the `sigma8_z` attribute can be adjusted to compute the clustering quantity at different redshifts.

Functions

The HZPT class objects are callable objects; they return either the HZPT power spectrum or correlation function at the specified wavenumber `k` or separation `r`. The `zeldovich` function returns the Zel'dovich term and the `broadband` function returns the Padé correction term.

Examples

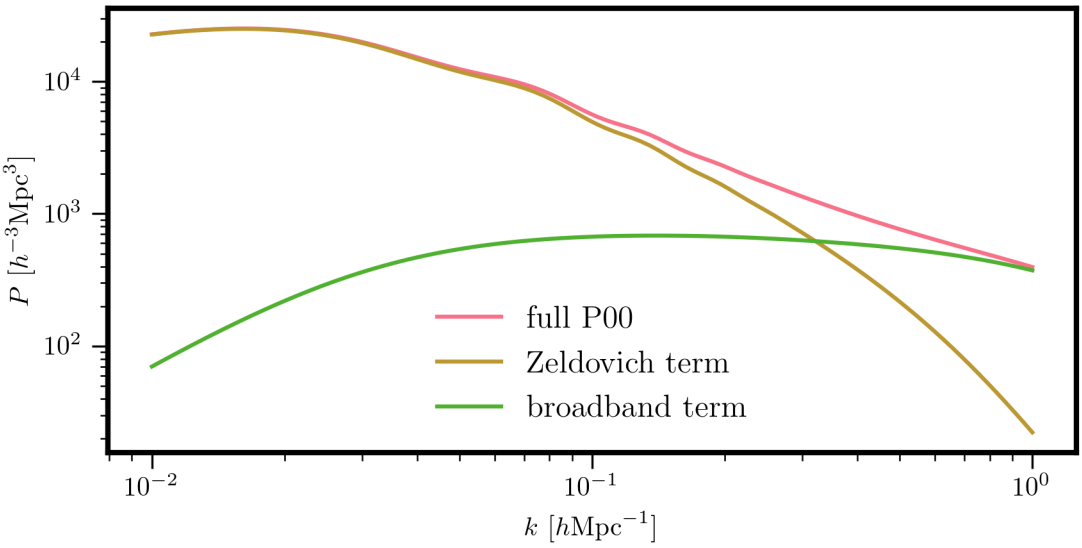
For example, to compute the dark matter auto spectrum for the Planck 2015 cosmology,

```
from pyRSD.rsd.cosmology import Planck15
from pyRSD.rsd.hzpt import HaloZeldovichP00

# power spectrum at z = 0
P00 = HaloZeldovichP00(Planck15, z=0.)

# compute the full power and each term
k = np.logspace(-2, 0, 100)
Pk = P00(k)
Pzel = P00.zeldovich(k)
Pbb = P00.broadband(k)
```

(continues on next page)

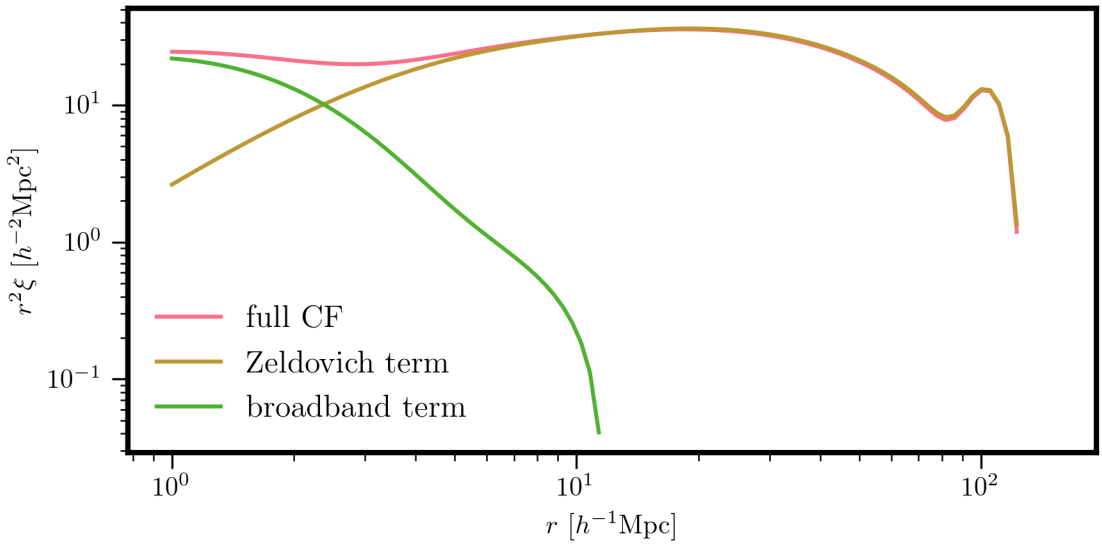
(continued from previous page)

```
# and plot
plt.loglog(k, Pk, label='full P00')
plt.loglog(k, Pzel, label='Zeldovich term')
plt.loglog(k, Pbb, label='broadband term')
```



Similary, the dark matter correlation function and the various terms can be computed as:

```
from pyRSD.rsd.hzpt import HaloZeldovichCF00

# correlation function at z = 0
CF = HaloZeldovichCF00(Planck15, z=0.)

# compute the full correlation and each term
r = np.logspace(0, np.log10(150), 100)
xi = CF(r)
xi_zel = CF.zeldovich(r)
xi_bb = CF.broadband(r)

# and plot
plt.loglog(r, r**2 * xi, label='full CF')
plt.loglog(r, r**2 * xi_zel, label='Zeldovich term')
plt.loglog(r, r**2 * xi_bb, label='broadband term')
```



1.6.2 API

The power spectrum classes are:

HaloZeldovichP00(cosmo, z)	The dark matter auto-spectrum P00 using Halo-Zel'dovich Perturbation Theory
HaloZeldovichP01(cosmo, z)	The dark matter density - radial momentum cross correlation P01 using HZPT
HaloZeldovichP11(cosmo, z)	The *mu4* contribution to the radial momentum auto spectrum P11, using HZPT
HaloZeldovichPhm(cosmo, z)	The halo-matter cross-correlation Phm, using HZPT

Each of these objects provides three main functions to compute the various HZPT terms:

__call__(k)	Return the total power at the specified k
zeldovich(k)	Return the Zel'dovich power term at the specified k
broadband(k)	The broadband power in units of $(\mathrm{Mpc}/h)^3$

The correlation function classes are:

HaloZeldovichCF00(cosmo, z)	The dark matter correlation function using Halo-Zel'dovich Perturbation Theory
HaloZeldovichCFhm(cosmo, z)	The dark matter - halo correlation function using Halo-Zel'dovich

Similary, the three main functions to compute the various HZPT terms are:

__call__(r)	Return the total correlation function
zeldovich(r)	Return the Zel'dovich correlation at the specified r
broadband(r)	The correlation function broadband correction term

HZPT Classes

class pyRSD.rsd.hzpt.**HaloZeldovichP00**(*cosmo*, *z*)

The dark matter auto-spectrum P00 using Halo-Zel'dovich Perturbation Theory

__call__(*k*)

Return the total power at the specified *k*

The total power is equal to the Zel'dovich power + broadband term

Parameters **k** : float, array_like

the wavenumber in units of h/Mpc

__init__(*cosmo*, *z*)

Parameters **cosmo** : cosmology.Cosmology, pygcl.Cosmology

the cosmology instance

z : float

the redshift; this determines the values of sigma8(z) to use in the HZPT equations

broadband(*k*)

The broadband power in units of $(\mathrm{Mpc}/h)^3$

The functional form is given by:

$$P_{\mathrm{BB}} = A_0 F(k) \left[\frac{1 + (kR_1)^2}{1 + (kR_{1h})^2 + (kR_{2h})^4} \right],$$

as given by Eq. 1 in arXiv:1501.07512.

Parameters **k** : float, array_like

the wavenumber in units of h/Mpc

zeldovich(*k*)

Return the Zel'dovich power term at the specified *k*

Parameters **k** : float, array_like

the wavenumber in units of h/Mpc

class pyRSD.rsd.hzpt.**HaloZeldovichP01**(*cosmo*, *z*)

The dark matter density - radial momentum cross correlation P01 using HZPT

__call__(*k*)

Return the full Halo Zeldovich P01

__init__(*cosmo*, *z*)

Parameters **cosmo** : cosmology.Cosmology, pygcl.Cosmology

the cosmology instance

z : float

the redshift; this determines the values of sigma8(z) to use in the HZPT equations

broadband(*k*)

The broadband power correction for P01 in units of (Mpc/h)^3

This is the derivative of the broadband band term for P00, taken with respect to `lna`

zeldovich(*k*)

Return the Zel'dovich power term at the specified *k*

Parameters **k** : float, array_like

the wavenumber in units of h/Mpc

class pyRSD.rsd.hzpt.**HaloZeldovichP11**(*cosmo*, *z*)

The *mu4* contribution to the radial momentum auto spectrum P11, using HZPT

Notes

The 1-loop SPT model for the vector contribution to P11[mu4] should be added to the power returned by this class to model the full P11[mu4]

__call__(*k*)

Return the total power

__init__(*cosmo*, *z*)

Parameters **cosmo** : cosmology.Cosmology, pygcl.Cosmology

the cosmology instance

z : float

the redshift; this determines the values of sigma8(z) to use in the HZPT equations

broadband(*k*)

The broadband power correction in units of (Mpc/h)^3

Modeled with a Pade function,

$$P_{BB} = F(k)A_0/(1 + (kR_{1h})^2)$$

zeldovich(*k*)

Return the Zel'dovich power term at the specified *k*

Parameters **k** : float, array_like

the wavenumber in units of h/Mpc

class pyRSD.rsd.hzpt.**HaloZeldovichPhm**(*cosmo*, *z*)

The halo-matter cross-correlation Phm, using HZPT

__call__(*b1*, *k*)

Return the total power, equal to the b1 * Zeldovich power + broadband correction

Parameters **b1** : float

the linear bias to compute the bias at

k : array_like

the wavenumbers in h/Mpc to compute the power at

__init__(*cosmo*, *z*)

Parameters **cosmo** : cosmology.Cosmology, pygcl.Cosmology

the cosmology instance

z : float

the redshift; this determines the values of sigma8(z) to use in the HZPT equations

broadband(*k*)
The broadband power in units of $(\mathrm{Mpc}/h)^3$

The functional form is given by:

$$P_{\rm BB} = A_0 F(k) \left[\frac{1 + (kR_1)^2}{1 + (kR_{1h})^2 + (kR_{2h})^4} \right],$$

as given by Eq. 1 in arXiv:1501.07512.

Parameters **k** : float, array_like

the wavenumber in units of h/Mpc

zeldovich(*k*)
Return the Zel'dovich power term at the specified *k*

Parameters **k** : float, array_like

the wavenumber in units of h/Mpc

class pyRSD.rsd.hzpt.**HaloZeldovichCF00**(*cosmo*, *z*)
The dark matter correlation function using Halo-Zel'dovich Perturbation Theory

__call__(*r*)
Return the total correlation function

__init__(*cosmo*, *z*)

Parameters **cosmo** : cosmology.Cosmology, pygcl.Cosmology

the cosmology instance

z : float

the redshift; this determines the values of sigma8(z) to use in the HZPT equations

broadband(*r*)
The correlation function broadband correction term

This is given by the Fourier transform of the Pade function

Parameters **r** : float, array_like

the separation array, in units of Mpc/h

zeldovich(*r*)
Return the Zel'dovich correlation at the specified *r*

Parameters **r** : float, array_like

the separation array, in units of Mpc/h

class pyRSD.rsd.hzpt.**HaloZeldovichCFhm**(*cosmo*, *z*)
The dark matter - halo correlation function using Halo-Zel'dovich Perturbation Theory

__call__(*b1*, *r*)
Return the total correlation

Parameters **b1** : float

the linear bias to compute correlation at

r : array_like

the separations in Mpc/h to correlation at

__init__(*cosmo*, *z*)

Parameters **cosmo** : cosmology.Cosmology, pygcl.Cosmology

the cosmology instance

z : float

the redshift; this determines the values of sigma8(z) to use in the HZPT equations

broadband(*r*)
The correlation function broadband correction term

This is given by the Fourier transform of the Pade function

zeldovich(*r*)
Return the Zel'dovich correlation at the specified *r*

1.7 Galaxy Power Spectrum

The *pyRSD.rsd.GalaxySpectrum* class computes the redshift-space galaxy power spectrum.

1.7.1 Overview

Theoretical Background

The model is described in detail in the paper Hand et al. 2017, but we summarize the relevant information needed to get started using pyRSD below.

Our galaxy model decomposes the galaxy sample into several subsamples, and computes the correlations between each of those subsamples. The galaxy sample gets divided into four subsamples:

Sample	Description
type A centrals	isolated centrals (no satellites in the same halo)
type B centrals	non-isolated centrals (at least one satellite in same halo)
type A satellites	isolated satellites (no other satellites in same halo)
type B satellites	non-isolated satellites (at least one other satellite in the same halo)

This sample decomposition is useful because it allows us to separately model the correlations between galaxies in the same halo (denoted as "1-halo" terms) and correlations that arise from galaxies in separate dark matter halos (denoted as "2-halo" terms).

Thus, we can model the total galaxy power spectrum as

$$P^{gg}(k,\mu) = (1-f_s)^2 P^{cc} + 2f_s(1-f_s)P^{cs} + f_s^2 P^{ss},$$

where f_s is the satellite fraction and P^{cc}, P^{cs}, and P^{ss} are the centrals auto power, central-satellite cross power, and satellite auto power, respectively.

Model Initialization

The galaxy power spectrum model can be initialized from a cosmology and a redshift specified by the user. For example, you can initialize the model at $z = 0$ for the Planck 2015 cosmology parameters as

```
In [1]: from pyRSD.rsd import GalaxySpectrum

In [2]: from pyRSD.rsd.cosmology import Planck15

In [3]: model = GalaxySpectrum(z=0, params=Planck15)
```

Once the model object has been created, the underlying elements of the model need to be initialized. Depending on your machine, this will take several minutes to complete.

```
# model takes several minutes to initialize once
model.initialize()

# set kmin/kmax limits
model.kmin = 1e-3
model.kmax = 0.5
```

The initialization only needs to be done once, and once the model initialized, the evaluation of the model will be fast (typically < 1 second), as long as the desired k value is valid, as specified by the `kmin` and `kmax` attributes of the model.

Warning: The *GalaxySpectrum* class has attributes `kmin` and `kmax` that specify where the valid wavenumber range over which the underlying model has been initialized. Outside of this range, the model must evaluate each component of the model for each k value, which can be time-consuming. A warning will be printed when this occurs.

Because the model initialization is time consuming, we recommend saving the initialized model and then reading the model from disk when it is needed again. This can be achieved with the numpy's pickling functionality:

```
# save the initialized model to disk
model.to_npy('galaxy_model.npy')

# read a new model from disk
model2 = GalaxySpectrum.from_npy('galaxy_model.npy')
```

Model Parameters

The GalaxySpectrum object has several model parameters. These parameters all of default values, but the attributes can be explicitly set by the user to change the behavior of the model.

The parameters are:

Cosmology	Name	Description
$\alpha_{\parallel}$	alpha_par	The Alcock-Paczynski effect parameter for parallel to the line-of-sight
$\alpha_{\perp}$	alpha_perp	The Alcock-Paczynski effect parameter for perpendicular to the line-of-sight
f	f	The growth rate at z: $f = d\ln D / d\ln a$
$\sigma_8(z)$	sigma8_z	The mass variance at $r = 8$ Mpc$/h$ at z; this sets the linear power spectrum normalization
Linear biases		
b_{1,c_A}	b1_cA	The linear bias of the type A centrals sample
b_{1,c_B}	b1_cB	The linear bias of the type B centrals sample
b_{1,s_A}	b1_sA	The linear bias of the type A satellites sample
b_{1,s_B}	b1_sB	The linear bias of the type B satellites sample
Sample fractions		
f_s	f_s	The satellite fraction, which is (total number of satellites / total number of galaxies)
f_{c_B}	f_cB	The type B centrals fraction, which is (total number of type B centrals / total number of centrals)
f_{s_B}	f_sB	The type B satellites fraction, which is (total number of type B satellites / total number of satellites)
Finger-of-God damping		
σ_c	sigma_c	The centrals FoG velocity dispersion, in units of Mpc$/h$
σ_{s_A}	sigma_sA	The type A satellites FoG velocity dispersion, in units of Mpc$/h$
σ_{s_B}	sigma_sB	The type B satellites FoG velocity dispersion, in units of Mpc$/h$
1-halo amplitudes		
$N_{c_B s}$	N_cBs	The amplitude of the constant 1-halo term between type B centrals, [units: $(\mathrm{Mpc}/h)^3$]
$N_{s_B s_B}$	N_sBsB	The amplitude of the constant 1-halo term between type B satellites, [units: $(\mathrm{Mpc}/h)^3$]

The parameters can be updated via the usual attribute setting mechanism

```
# update normalization via sigma8_z
In [4]: model.sigma8_z = 0.62

# update satellite fraction
In [5]: model.f_s = 0.12
```

1.7.2 Computing Power Spectra

With an initialized model, we can compute the power spectrum either as a function of k and μ, $P(k, \mu)$, or compute the multipoles of the power spectrum, $P_\ell(k)$. This is accomplished either by calling the *GalaxySpectrum.power()* or *GalaxySpectrum.poles()* functions.

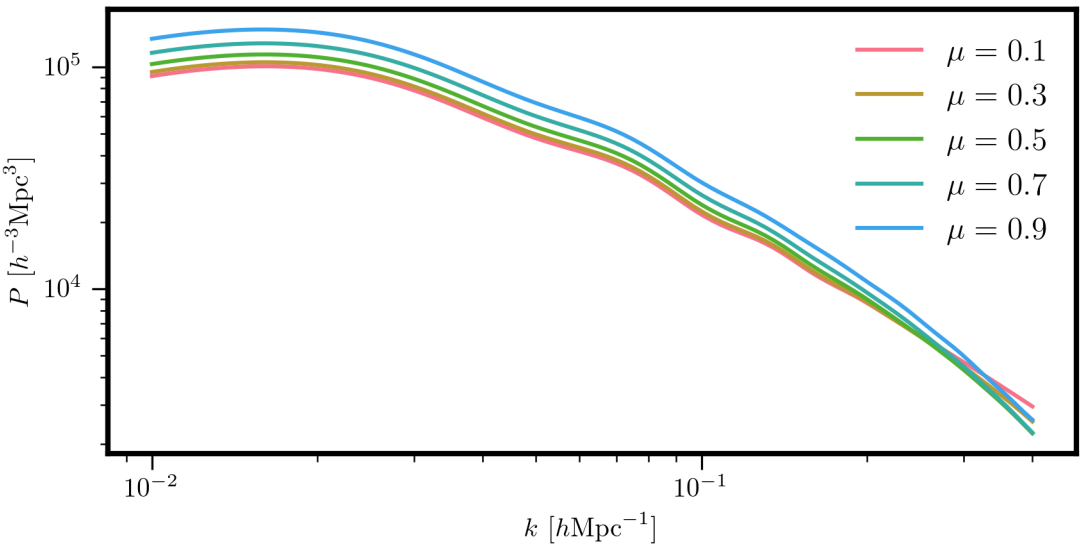
For example, to compute $P(k, \mu)$ for 5 μ bins:

```
k = numpy.logspace(-2, numpy.log10(0.4), 100)

# this is mu = 0.1, 0.3, 0.5, 0.7, 0.9
mu = numpy.arange(0.1, 1.0, 0.2)
Pkmu = model.power(k, mu) # shape is (100,5)
```

(continues on next page)

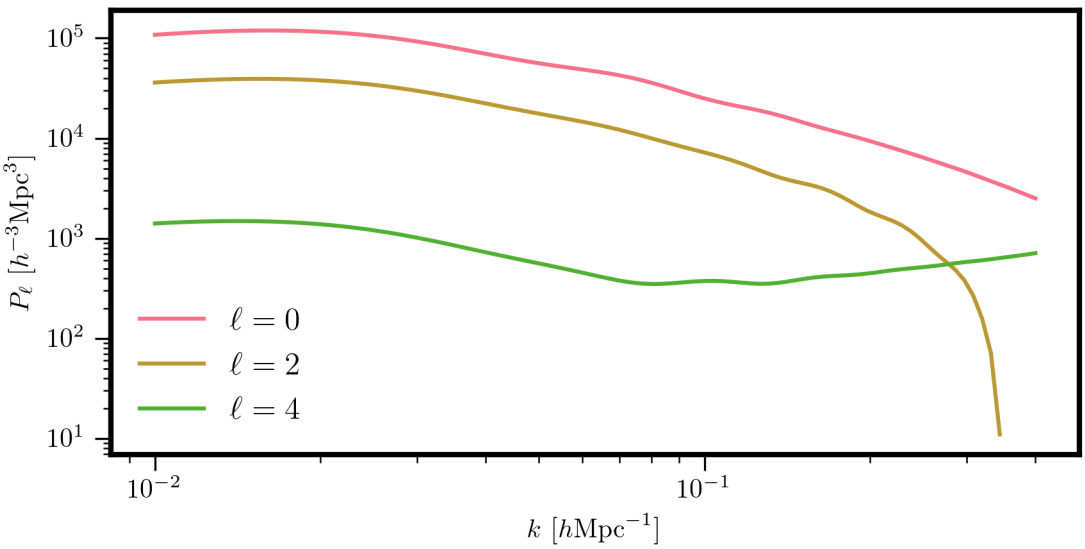
(continued from previous page)

```
for i, imu in enumerate(mu):
  plt.loglog(k, Pkmu[:,i], label=r"$\mu = %.1f$" %imu)
```



And, for example, the monopole, quadrupole, and hexadecapole ($\ell = 0, 2, 4$) can be computed as

```
ells = [0, 2, 4]
Pell = model.poles(k, ells) # list of 3 (100,) arrays

for i, ell in enumerate(ells):
  plt.loglog(k, Pell[i], label=r"$\ell = %d$" %ell)
```



1.7.3 Discrete Binning Effects

When computing the power spectrum from galaxy survey data, we estimate the power in bandpowers, averaging over a finite set of modes in each k or μ bin. Especially on large scales (low k), where we are averaging over relatively few values, this averaging procedure can lead to additional effects on the power spectrum. The pyRSD package accounts for these effects by applying an additional transfer function to the continuous power spectrum that is returned by `GalaxySpectrum.power()` or `GalaxySpectrum.poles()`.

The discretely-binned power spectrum theory is computed by either the `PkmuTransfer` or `PolesTransfer` classes, depending if the user wants $P(k, \mu)$ or $P_\ell(k)$. A transfer function can be initialized, and then the discretely-binned power spectrum theory will be returned by the `GalaxySpectrum.from_transfer()` function.

Specifying the (k, μ) Grid

In order to account for the discrete binning effects, the user must supply the average k and μ values on the 2D binning grid, as well as the number of modes averaged over in each bin. These bins should be finely spaced, typically about ~100 μ bins yields good results. The continuous theory will be evaluted for each these discrete bins, and then averaged over, weighted by the number of modes in each bin, to account for any binning effects.

The class that handles setting up the $P(k, \mu)$ grid is `PkmuGrid`. This class can be initialized

```
# set up fake 1D k, mu bins
In [1]: k_1d = numpy.arange(0.01, 0.4, 0.005)

In [2]: mu_1d = numpy.linspace(0, 1.0, 100)

# convert to a 2D grid
# shape is (78, 100)
In [3]: k, mu = numpy.meshgrid(k_1d, mu_1d, indexing='ij')

# assign random weights for each bin for illustration purposes
In [4]: modes = numpy.random.random(size=k.shape)

# simulate missing data
In [5]: missing = numpy.random.randint(0, numpy.prod(modes.shape), size=10)

In [6]: modes.flat[missing] = numpy.nan

# initialize the grid
In [7]: grid = PkmuGrid([k_1d, mu_1d], k, mu, modes)
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-7-235933c43c91> in <module>()
----> 1 grid = PkmuGrid([k_1d, mu_1d], k, mu, modes)

NameError: name 'PkmuGrid' is not defined

In [8]: print(grid)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-8-8e1381097b77> in <module>()
----> 1 print(grid)

NameError: name 'grid' is not defined
```

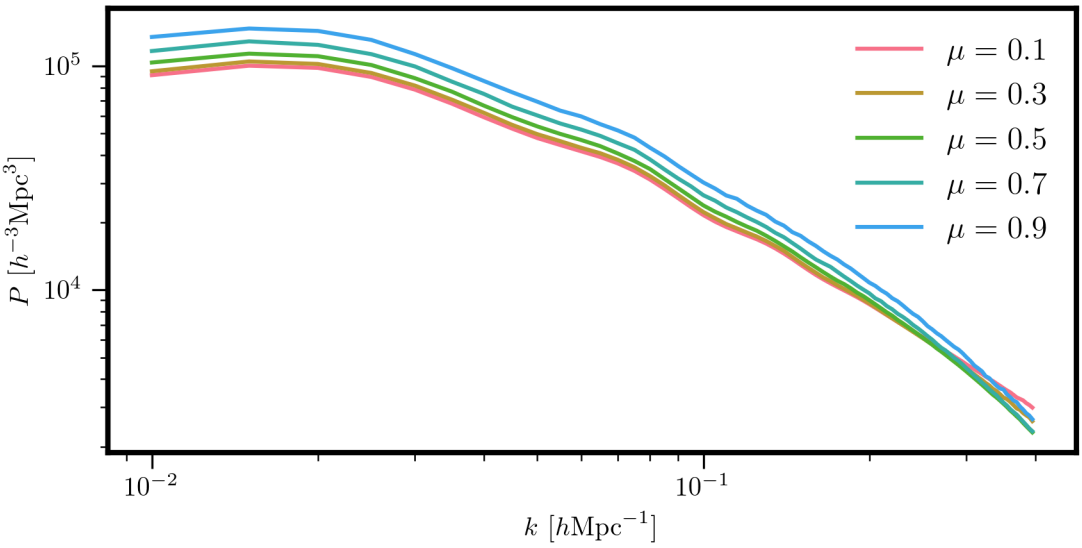
Once a `PkmuGrid` object has been initialized, we can save it to disk as a plaintext file for later use.

```
# save to a plaintext file
In [9]: grid.to_plaintext('pkmu_grid.dat')
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-9-f4b63e7a5632> in <module>()
----> 1 grid.to_plaintext('pkmu_grid.dat')

NameError: name 'grid' is not defined

# read the grid back from disk
In [10]: grid_2 = PkmuGrid.from_plaintext('pkmu_grid.dat')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-10-7de7eb7ca8d9> in <module>()
----> 1 grid_2 = PkmuGrid.from_plaintext('pkmu_grid.dat')

NameError: name 'PkmuGrid' is not defined

In [11]: print(grid_2)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-11-fb6d86bbdf3d> in <module>()
----> 1 print(grid_2)

NameError: name 'grid_2' is not defined
```

Discrete $P(k, \mu)$

Once the binning grid is initialized, we next must initialize the desired transfer function. Imagine we want to compute the power spectrum in 5 μ bins, accounting for discrete binning effects. This can be achieved as

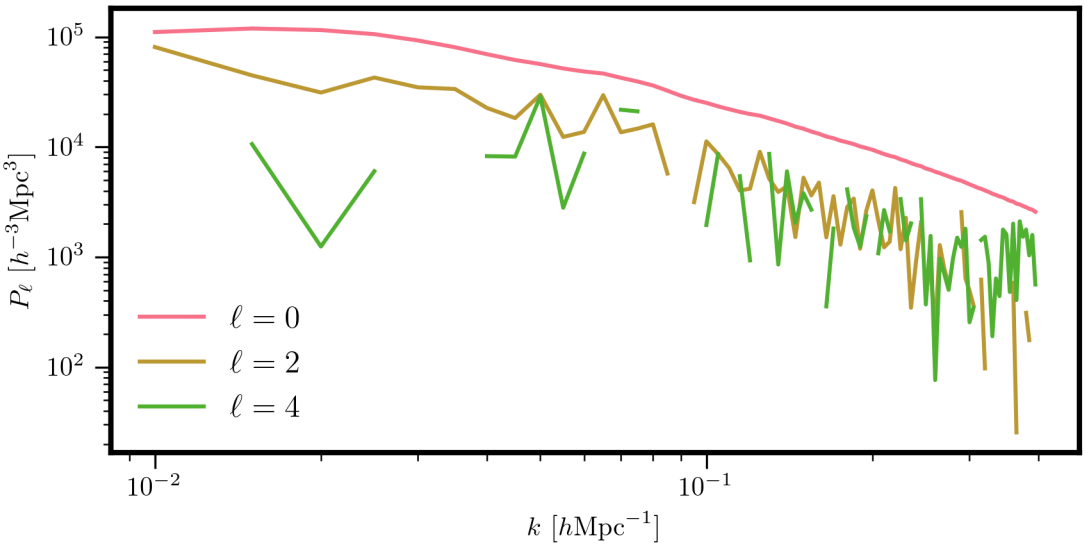
```
# edges of the mu bins
mu_bounds = [(0., 0.2), (0.2, 0.4), (0.4, 0.6), (0.6, 0.8), (0.8, 1.0)]

# the transfer function, with specified valid k range
transfer = PkmuTransfer(grid, mu_bounds, kmin=0.01, kmax=0.4)

# evaluate the model with this transfer function
Pkmu_binned = model.from_transfer(transfer)

# get the coordinate arrays from the grid
k, mu = transfer.coords # this has shape of (Nk, Nmu)

for i in range(mu.shape[1]):
  plt.loglog(k[:,i], Pkmu_binned[:,i], label=r"$\mu = %.1f$" % mu[:,i].mean())
```



Discrete $P_\ell(k)$

We can similarly compute multipoles while accounting for discrete binning effects. This can be achieved as

```
# the multipoles to compute
ells = [0, 2, 4]

# the transfer function, with specified valid k range
transfer = PolesTransfer(grid, ells, kmin=0.01, kmax=0.4)

# evaluate the model with this transfer function
poles_binned = model.from_transfer(transfer) # shape is (78, 3)

# get the coordinate arrays from the grid
k, mu = transfer.coords # this has shape of (Nk, Nmu)

for i, iell in enumerate(ells):
  plt.loglog(k[:,i], poles_binned[:,i], label=r"$\ell = %d$" % iell)
```



1.7.4 Window-convolved Power Spectra

When analyzing power spectrum measurements from a galaxy survey, the theoretical quantity of interest is the power spectrum convolved with the survey window function, which arises due to the presence of the survey selection function. In particular, the most common power spectrum spectrum from galaxy survey data is the window-convolved power spectrum multipoles. See 1704.02357 for more details about how this measurement is made from survey data.

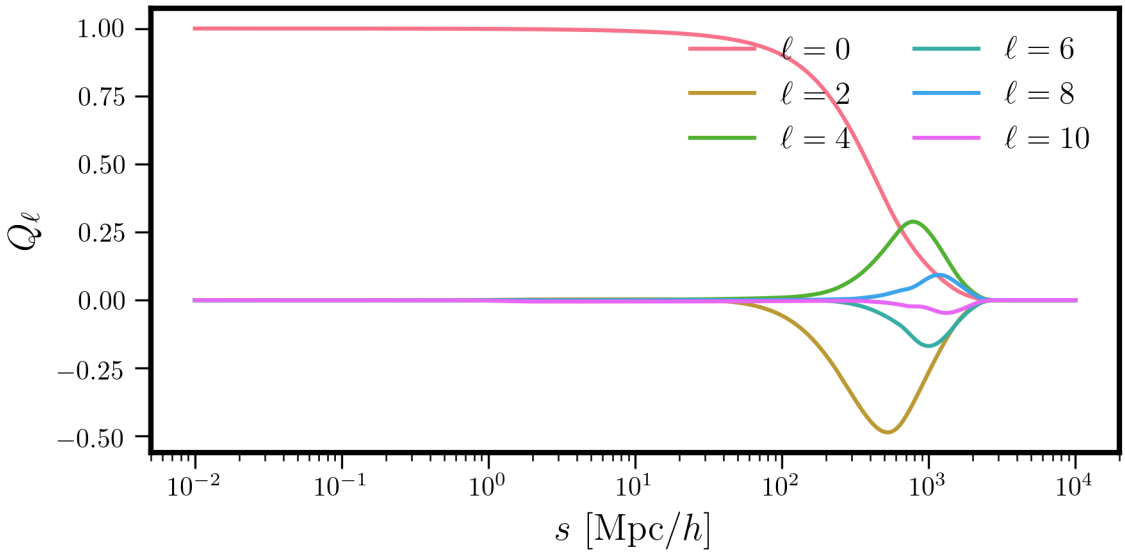
The pyRSD package includes functionality for computing the theoretical power spectrum multipoles convolved with a window function input by the user. The window convolution procedure implemented in pyRSD is described in detail in Wilson et al. 2017, as well as in Section 4.3.2 of 1706.02362. Please see these references for further details about the algorithm.

Specifying the Window Function

In order to convolve the theoretical power spectrum multipoles with a survey window function, the user must input the correlation function multipoles of the survey window in configuration space. Specifically, for a set of separations s, the user should supply the window multipoles $Q_\ell(s)$ for $\ell = 0, 2, 4, ..., \ell_{\mathrm{max}}$. In order to fully account for the anisotropy introduced by the window function on the model, we recommend that $\ell_{\mathrm{max}} = 8$ or $\ell_{\mathrm{max}} = 10$, to be cautious.

Below, we show the window function multipoles in configuration space as measured with a pair counter correlation function code for the BOSS DR12 CMASS sample.

```
# load the window function correlation multipoles array from disk
# first column is s, followed by W_ell
Q = numpy.loadtxt('data/window.dat')

# now plot
for i in range(1, Q.shape[1]):
  plt.semilogx(Q[:,0], Q[:,i], label=r"$\ell=%d$" %(2*(i-1)))
```

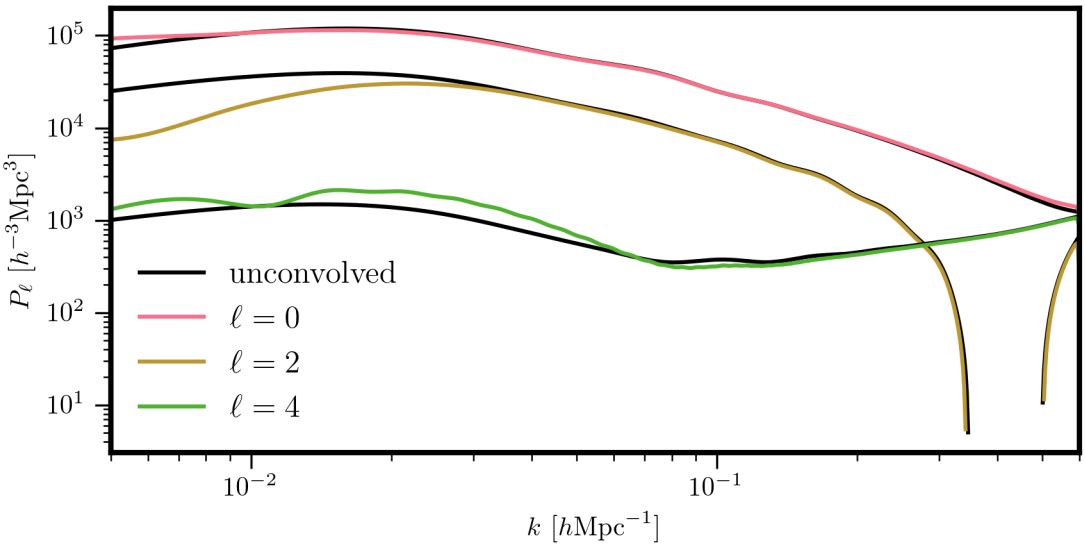


From this plot we can see that the Q_ℓ vanish for scales approaching 3000 Mpc/h, as these are the largest scales in the volume of the CMASS sample. Also note that on small scales, the clustering becomes isotropic, with the multipoles vanishing, and that in general, the contribution of the higher-order multipoles decreases as ℓ increases.

Evaluating the Convolution

Once the window multipoles have been measured, we can evaluate the convolved multipoles with the transfer function class, `WindowTransfer`. For example, to evalute the convolved monopole, quadrupole, and hexadecapole, we simply do

```
# adjust the model kmin/kmax
model.kmin = 1e-4
model.kmax = 0.7

# the multipoles to compute
ells = [0, 2, 4]

# the window transfer function, with specified valid k range
transfer = WindowTransfer(Q, ells, grid_kmin=1e-3, grid_kmax=0.6)

# evaluate the model with this transfer function
Pell_conv = model.from_transfer(transfer) # shape is (78, 3)

# get the coordinate arrays from the grid
k, mu = transfer.coords # this has shape of (Nk, Nmu)

for i, iell in enumerate(ells):
  plt.loglog(k[:,i], Pell_conv[:,i], label=r"$\ell = %d$" % iell)
```



In this plot, we show the unconvoled P_0, P_2, and P_4 multipoles in black, and the correpsonding window-convolved multipoles in color. The effects of the window function, mostly on large scales (small k) are clearly evident.

Warning: The convolution is performed in configuration space using the Convolution Theorem. FFTLog is used to perform the Fourier transfrom on a grid, and to achieve the best results, the grid bounds should be padded to cover a wider range than the wavenumbers of interest. Typically, if one is interested in the power at $k = 0.4\ h/\mathrm{Mpc}$, the `grid_kmax` keyword should be set to $k = 0.6\ h/\mathrm{Mpc}$ or $k = 0.7\ h/\mathrm{Mpc}$, as was done in the example above. Also, note that to avoid slowdowns, the model `kmin` and `kmax` attributes should be adjusted to include the desired grid range, as was done above.

1.7.5 API

Top level user functions to compute power spectra:

GalaxySpectrum([fog_model, use_so_correction])	The model for the galaxy redshift space power spectrum
GalaxySpectrum.initialize()	Initialize the underlying splines, etc of the model
GalaxySpectrum.power(k, mu[, flatten])	The total redshift-space galaxy power spectrum at `k` and `mu`
GalaxySpectrum.poles(k, ells[, Nmu])	The multipole moments of the redshift-space power spectrum
GalaxySpectrum.from_transfer	

Loading GalaxySpectrum objects from and saving to pickle files:

GalaxySpectrum.to_npy(filename)	Save to a `.npy` file by calling `numpy.save()`
GalaxySpectrum.from_npy(filename)	Load a model from a `.npy` file

Generating the default set of parameters for fitting the *GalaxySpectrum* to data:

GalaxySpectrum.default_params()	A GalaxyPowerParameters object holding the default model parameters

Evaluating power spectra with an additional transfer function, i.e., on a discrete (k, mu) grid or convolved with a window:

PkmuGrid
PkmuTransfer
PolesTransfer
pyRSD.rsd.window.WindowTransfer

The GalaxySpectrum Class

class pyRSD.rsd.**GalaxySpectrum**(*fog_model='modified_lorentzian'*, *use_so_correction=False*, ***kwargs*)

The model for the galaxy redshift space power spectrum

Parameters **kmin** : float, optional

The minimum wavenumber to compute the power spectrum at [units: h/Mpc]; default is 1e-3

kmax : float, optional

The maximum wavenumber to compute the power spectrum at [units: h/Mpc]; default is 0.5

Nk : int, optional

The number of log-spaced bins to use as the underlying domain for splines; default is 200

z : float, optional

The redshift to compute the power spectrum at. Default = 0.

params : *Cosmology*, str

Either a *Cosmology* instance or the name of a file to load parameters from; see the 'data/params' directory for examples

include_2loop : bool, optional

If *True*, include 2-loop contributions in the model terms. Default is *False*.

transfer_fit : str, optional

The name of the transfer function fit to use. Default is *CLASS* and the options are {*CLASS*, *EH*, *EH_NoWiggle*, *BBKS*}, or the name of a data file holding (k, T(k))

max_mu : {0, 2, 4, 6, 8}, optional

Only compute angular terms up to mu**(max_mu). Default is 4.

interpolate: bool, optional

Whether to return interpolated results for underlying power moments

k0_low : float, optional (*5e-3*)

below this wavenumber, evaluate any power in "low-k mode", which essentially just uses SPT at low-k

linear_power_file : str, optional (*None*)

string specifying the name of a file which gives the linear power spectrum, from which the transfer function in `cosmo` will be initialized

Pdv_model_type : {'jennings', 'sim', None}, optional

The type of model to use to evaluate Pdv

fog_model : str, optional

the string specifying the FOG model to use; one of ['modified_lorentzian', 'lorentzian', 'gaussian']. Default is 'modified_lorentzian'

use_so_correction : bool, optional

Boost the centrals auto spectrum with a correction accounting for extra structure around centrals due to SO halo finders; default is *False*

power(*k*, *mu*, *flatten=False*)

The total redshift-space galaxy power spectrum at `k` and `mu`

Parameters **k** : float, array_like

The wavenumbers to evaluate the power spectrum at, in *h/Mpc*

mu : float, array_like

The cosine of the angle from the line of sight. If a float is provided, the value is used for all input *k* values. If array-like and *mu* has the same shape as *k*, the power at each (k,mu) pair is returned. If *mu* has a shape different than *k*, the returned power has shape `(len(k), len(mu))`.

flatten : bool, optional

If *True*, flatten the return array, which will have a length of *len(k) * len(mu)*

default_params()

A GalaxyPowerParameters object holding the default model parameters

The model associated with the parameter is `self`

poles(*k*, *ells*, *Nmu=40*)

The multipole moments of the redshift-space power spectrum

Parameters **k** : float, array_like

The wavenumbers to evaluate the power spectrum at, in *h/Mpc*

ells : int, array_like

The *ell* values of the multipole moments

Nmu : int, optional

the number of `mu` bins to use when performing the multipole integration

Returns **poles** : array_like

returns tuples of arrays for each ell value in `poles`

Gridded Power Spectra

Window-convolved Power Spectra

1.8 Quasar Power Spectrum

This page is under construction.

RSDFit

The **RSDFit** module deals with running parameter estimation using the power spectrum models available in this package and data input by the user.

- *Getting Started*
- *Specifying the Data*
- *Specifying the Theory*
- *Running the Fits*
- *Exploring the Results*
- *Advanced Patterns*

1.9 Getting Started

1.9.1 The `rsdfit` Executable

The pyRSD package includes a `rsdfit` executable that is capable of running parameter estimation using the power spectrum models *`GalaxySpectrum`* and `QuasarSpectrum` to fit power spectrum data input by the user.

After installing the pyRSD package, the `rsdfit` command should be located on your `PATH`. We can then print the help message for the command via

```
$ rsdfit -h
usage: rsdfit [-h] [--version] {mcmc,nlopt,restart,analyze} ...

From more help on each of the subcommands, type:
rsdfit mcmc -h
rsdfit nlopt -h
rsdfit restart -h
rsdfit analyze -h

fitting redshift space power spectrum observations with the `pyRSD` model

positional arguments:
  {mcmc,nlopt,restart,analyze}
    mcmc                find the best-fitting parameters using emcee to run
                        MCMC chains
    nlopt               use nonlinear optimization to find the best-fitting
                        using the LBFGS algorithm
    restart             restart a parameter estimation fit from an existing
                        result
    analyze             analyze the results of an MCMC parameter fit

optional arguments:
```

(continues on next page)

(continued from previous page)

```
  -h, --help            show this help message and exit
  --version             print the pyRSD version and exit
```

The `rsdfit` command contains four sub-commands: **nlopt**, **mcmc**, **restart**, and **analyze**. The most important of these sub-commands are `nlopt` and `mcmc`. This allows the user to run fresh parameter fits from scratch, which is the main use case for the `rsdfit` executable.

The calling sequence for the `mcmc` command is

```
$ rsdfit mcmc -h
usage: rsdfit [-h] [--version] {mcmc,nlopt,restart,analyze} ...

From more help on each of the subcommands, type:
rsdfit mcmc -h
rsdfit nlopt -h
rsdfit restart -h
rsdfit analyze -h mcmc
       [-h] [-m MODEL] [-p PARAMS] [--silent] -w WALKERS -i ITERATIONS
       [-n NCHAINS] -o FOLDER [--debug] [--no-save-model]

optional arguments:
  -h, --help            show this help message and exit
  -m MODEL, --model MODEL
                        file name holding the model path
  -p PARAMS, --params PARAMS
                        file name holding the driver, theory, and data
                        parameters
  --silent              silence the standard output to the console
  -w WALKERS            number of emcee walkers to run the MCMC chain
                        (required)
  -i ITERATIONS         number of steps to run in the MCMC chain (required)
  -n NCHAINS, --nchains NCHAINS
                        number of MCMC chains to run concurrently
  -o FOLDER, --output FOLDER
                        the folder where the results will be written
                        (required)
  --debug               whether to print more info about the mpi4py.Pool
                        object
  --no-save-model       do not save the model instance
```

and for the `nlopt` command is

```
$ rsdfit nlopt -h
usage: rsdfit [-h] [--version] {mcmc,nlopt,restart,analyze} ...

From more help on each of the subcommands, type:
rsdfit mcmc -h
rsdfit nlopt -h
rsdfit restart -h
rsdfit analyze -h nlopt
       [-h] [-m MODEL] [-p PARAMS] [--silent] -i ITERATIONS -o FOLDER
       [--debug] [--no-save-model]

optional arguments:
  -h, --help            show this help message and exit
  -m MODEL, --model MODEL
                        file name holding the model path
```

(continues on next page)

(continued from previous page)

```
  -p PARAMS, --params PARAMS
                        file name holding the driver, theory, and data
                        parameters
  --silent              silence the standard output to the console
  -i ITERATIONS         the maximum number of iterations to run (required)
  -o FOLDER, --output FOLDER
                        the folder where the results will be written
                        (required)
  --debug               whether to print more info about the mpi4py.Pool
                        object
  --no-save-model       do not save the model instance
```

The three most important options for these sub-commands are:

1. **-p, —params**

   The name of the main parameter file to load; this holds all of the relevant information about the data, theory, and main driver parameters. We'll discuss these parameter files in more detail in the next section.

2. **-o, —output**

   This is the name of the directory where the results will be saved. Each time `rsdfit` is run, the parameter file is saved to a file "params.dat" in this directory as well. If this folder already exists and contains a "params.dat" file, those parameters will be used and results will be added to the existing directory.

3. **-m, —model**

   The name of a file holding a *GalaxySpectrum* or `QuasarSpectrum` object to be loaded by the code. This argument is optional, but providing an already initialized model file will save a significant amount of time, since the code won't need to initialize a model from scratch. See *Model Initialization* for more details on initializing and saving models.

1.9.2 Parameter Files

The `rsdfit` command is configured through a single main parameter file. This parameter file is responsible for not only configuring the main `rsdfit` executable, but also for specifying the relevant theory parameters and information about the data measurements to load. To separate the different classes of parameters, the parameter names are prefixed with an additional identifier, one of

1. **driver**: general parameters that determine the `rsdfit` configuration
2. **data**: the parameters specifying the data and covariance matrix to load
3. **theory**: the theoretical parameters for the desired pyRSD model, which determines the free pararameters, constrained parameters, etc, during the fitting procedure
4. **model**: parameters specifying the *GalaxySpectrum* or `QuasarSpectrum` configuration; these are parameters that are passed to the `__init__()` function of these model classes

We'll be exploring each of these parameter types in much more detail in the next few sections, but for now, let's take a quick look at an example parameter file:

```
#------------------------------------------------------------------------------
# driver_params
#------------------------------------------------------------------------------
driver.burnin = 0
```

(continues on next page)

(continued from previous page)

```
driver.epsilon = 0.02
driver.init_from = 'fiducial'
driver.init_scatter = 0.0
driver.lbfgs_epsilon = {'Nsat_mult': 0.01, 'f1h_cBs': 0.01}
driver.lbfgs_options = {'ftol': 1e-10, 'xtol': 1e-10, 'gtol': 1e-05}
driver.lbfgs_use_priors = True
driver.solver_type = 'nlopt'
driver.start_from = None
driver.test_convergence = False
#------------------------------------------------------------------------------

#------------------------------------------------------------------------------
# data params
#------------------------------------------------------------------------------
data.covariance = '$(PYRSD_DATA)/examples/runPB_poles_gaussian_cov.dat'
data.covariance_Nmocks = 0
data.covariance_rescaling = 1.0
data.data_file = '$(PYRSD_DATA)/examples/runPB_galaxy_poles.dat'
data.ells = [0, 2, 4]
data.fitting_range = [(0.02, 0.4), (0.02, 0.4), (0.02, 0.4)]
data.grid_file = '$(PYRSD_DATA)/examples/runPB_pkmu_grid.dat'
data.mode = 'poles'
data.mu_bounds = None
data.statistics = ['pole_0', 'pole_2', 'pole_4']
data.usedata = range(0, 3)
data.window_file = None
#------------------------------------------------------------------------------

#------------------------------------------------------------------------------
# theory params
#------------------------------------------------------------------------------
theory.F_AP = {'value': 1.0, 'vary': False, 'analytic': False, 'name': 'F_AP', 'expr
↪': 'alpha_par/alpha_perp'}
theory.N = {'value': 0, 'vary': False, 'analytic': False, 'min': 0, 'name': 'N',
↪'lower': 0, 'prior': 'uniform', 'fiducial': 0, 'upper': 500}
theory.NcBs = {'value': 29608.657627046545, 'vary': False, 'analytic': False, 'name':
↪'NcBs', 'fiducial': 45000.0, 'expr': 'f1h_cBs / (fcB*(1 - fs)*nbar)'}
theory.NsBsB = {'value': 94583.2118641765, 'vary': False, 'analytic': False, 'name':
↪'NsBsB', 'fiducial': 94500.0, 'expr': 'f1h_sBsB / (fsB**2 * fs**2 * nbar) * (fcB*(1␣
↪- fs) - fs*(1-fsB))'}
theory.Nsat_mult = {'value': 2.4, 'vary': True, 'sigma': 0.2, 'min': 2.0, 'name':
↪'Nsat_mult', 'prior': 'normal', 'fiducial': 2.4, 'mu': 2.4, 'analytic': False}
theory.alpha = {'value': 1.0, 'vary': False, 'analytic': False, 'name': 'alpha', 'expr
↪': '(alpha_perp**2 * alpha_par)**(1./3)'}
theory.alpha_par = {'value': 1.0, 'vary': True, 'analytic': False, 'name': 'alpha_par
↪', 'prior': 'uniform', 'fiducial': 1.0, 'upper': 1.2, 'lower': 0.8}
theory.alpha_perp = {'value': 1.0, 'vary': True, 'analytic': False, 'name': 'alpha_
↪perp', 'prior': 'uniform', 'fiducial': 1.0, 'upper': 1.2, 'lower': 0.8}
theory.b1 = {'value': 2.124276, 'vary': False, 'analytic': False, 'name': 'b1', 'expr
↪': '(1 - fs)*b1_c + fs*b1_s'}
theory.b1_c = {'value': 1.9981383928571428, 'vary': False, 'analytic': False, 'name':
↪'b1_c', 'expr': '(1 - fcB)*b1_cA + fcB*b1_cB'}
theory.b1_cA = {'value': 1.9, 'vary': True, 'analytic': False, 'name': 'b1_cA', 'prior
↪': 'uniform', 'fiducial': 1.9, 'upper': 2.5, 'lower': 1.2}
theory.b1_cB = {'value': 3.0028260869565218, 'vary': False, 'analytic': False, 'name
↪': 'b1_cB', 'fiducial': 2.84, 'expr': '(1-fsB)/(1+fsB*(1./Nsat_mult - 1)) * b1_sA +␣
↪ (1 - (1-fsB)/(1+fsB*(1./Nsat_mult - 1))) * b1_sB'}
```

(continues on next page)

(continued from previous page)

```
theory.b1_s = {'value': 3.2109999999999994, 'vary': False, 'analytic': False, 'name':
↪'b1_s', 'expr': '(1 - fsB)*b1_sA + fsB*b1_sB'}
theory.b1_sA = {'value': 2.755, 'vary': False, 'analytic': False, 'name': 'b1_sA',
↪'fiducial': 2.63, 'expr': 'gamma_b1sA*b1_cA'}
theory.b1_sB = {'value': 3.8949999999999996, 'vary': False, 'analytic': False, 'name
↪': 'b1_sB', 'fiducial': 3.62, 'expr': 'gamma_b1sB*b1_cA'}
theory.b1sigma8 = {'value': 1.29580836, 'vary': False, 'analytic': False, 'name':
↪'b1sigma8', 'expr': 'b1*sigma8_z'}
theory.delta_sigsA = {'value': 1.0, 'vary': False, 'sigma': 0.2, 'min': 0.0, 'name':
↪'delta_sigsA', 'mu': 1.0, 'prior': 'normal', 'fiducial': 1.0, 'analytic': False}
theory.delta_sigsB = {'value': 1.0, 'vary': False, 'sigma': 0.2, 'min': 0.0, 'name':
↪'delta_sigsB', 'mu': 1.0, 'prior': 'normal', 'fiducial': 1.0, 'analytic': False}
theory.epsilon = {'value': 0.0, 'vary': False, 'analytic': False, 'name': 'epsilon',
↪'expr': '(alpha_perp/alpha_par)**(-1./3) - 1.0'}
theory.f = {'value': 0.78, 'vary': True, 'analytic': False, 'name': 'f', 'prior':
↪'uniform', 'fiducial': 0.78, 'upper': 1.0, 'lower': 0.6}
theory.f1h_cBs = {'value': 1.0, 'vary': True, 'sigma': 0.75, 'min': 0, 'name': 'f1h_
↪cBs', 'mu': 1.0, 'prior': 'normal', 'fiducial': 1.0, 'analytic': False}
theory.f1h_sBsB = {'value': 4.0, 'vary': True, 'sigma': 1.0, 'min': 0.0, 'name': 'f1h_
↪sBsB', 'prior': 'normal', 'fiducial': 4.0, 'mu': 4.0, 'analytic': False}
theory.f_so = {'value': 0.0, 'vary': False, 'sigma': 0.02, 'name': 'f_so', 'mu': 0.04,
↪ 'prior': 'normal', 'fiducial': 0.0, 'analytic': False}
theory.fcB = {'value': 0.08898809523809524, 'vary': False, 'analytic': False, 'min':␣
↪0, 'name': 'fcB', 'max': 1, 'fiducial': 0.089, 'expr': 'fs / (1 - fs) * (1 + fsB*(1.
↪/Nsat_mult - 1))'}
theory.fs = {'value': 0.104, 'vary': True, 'analytic': False, 'min': 0.0, 'name': 'fs
↪', 'max': 1, 'prior': 'uniform', 'fiducial': 0.104, 'upper': 0.25, 'lower': 0.0}
theory.fsB = {'value': 0.4, 'vary': True, 'analytic': False, 'min': 0.0, 'name': 'fsB
↪', 'max': 1, 'prior': 'uniform', 'fiducial': 0.4, 'upper': 1.0, 'lower': 0.0}
theory.fsigma8 = {'value': 0.4758, 'vary': False, 'analytic': False, 'name': 'fsigma8
↪', 'expr': 'f*sigma8_z'}
theory.gamma_b1cB = {'value': 0.4, 'vary': False, 'sigma': 0.2, 'min': 0.0, 'name':
↪'gamma_b1cB', 'max': 1, 'prior': 'normal', 'fiducial': 0.4, 'mu': 0.4, 'analytic':␣
↪False}
theory.gamma_b1sA = {'value': 1.45, 'vary': True, 'sigma': 0.3, 'min': 1.0, 'name':
↪'gamma_b1sA', 'prior': 'normal', 'fiducial': 1.45, 'mu': 1.45, 'analytic': False}
theory.gamma_b1sB = {'value': 2.05, 'vary': True, 'sigma': 0.3, 'min': 1.0, 'name':
↪'gamma_b1sB', 'prior': 'normal', 'fiducial': 2.05, 'mu': 2.05, 'analytic': False}
theory.nbar = {'value': 0.0004235857693396528, 'vary': False, 'analytic': False, 'name
↪': 'nbar', 'fiducial': 0.0004235857693396528}
theory.sigma8_z = {'value': 0.61, 'vary': True, 'analytic': False, 'name': 'sigma8_z',
↪ 'prior': 'uniform', 'fiducial': 0.61, 'upper': 0.9, 'lower': 0.3}
theory.sigma_c = {'value': 1.0, 'vary': True, 'analytic': False, 'name': 'sigma_c',
↪'prior': 'uniform', 'fiducial': 1.0, 'upper': 3.0, 'lower': 0.0}
theory.sigma_sA = {'value': 3.5, 'vary': True, 'analytic': False, 'name': 'sigma_sA',
↪'prior': 'uniform', 'fiducial': 3.5, 'upper': 8.0, 'lower': 2.0}
theory.sigma_sB = {'value': 4.7072613620519554, 'vary': False, 'analytic': False,
↪'name': 'sigma_sB', 'prior': 'uniform', 'fiducial': 5, 'expr': 'sigma_sA * sigmav_
↪from_bias(sigma8_z, b1_sB) / sigmav_from_bias(sigma8_z, b1_sA)', 'lower': 3.0,
↪'upper': 10.0}
theory.sigma_so = {'value': 0.0, 'vary': False, 'analytic': False, 'name': 'sigma_so',
↪ 'lower': 1.0, 'prior': 'uniform', 'fiducial': 0.0, 'upper': 7}
#------------------------------------------------------------------------------

theory_extra.b2_00__0 =  'A0 for generic b2_00'
theory_extra.gamma_b1sA =  'the relative fraction of b1_sA to b1_cA'
theory_extra.delta_sigsA =  'the relative fraction of sigma_sA to sigma_s'
```

(continues on next page)

(continued from previous page)

```
theory_extra.nbar =  'the mean number density in (h/Mpc)^3'
theory_extra.Nsat_mult =  'the mean number of satellites in halos with >1 sat'
theory_extra.gamma_b1cB =  'the relative fraction of b1_cB, varying linear between b1_
↪sA and b1_s'
theory_extra.delta_sigsB =  'the relative fraction of sigma_sB to sigma_s'
theory_extra.f_nbar =  'fraction multiplying nbar'
theory_extra.f1h_sBsB =  'fraction multiplying 1-halo term, NsBsB'
theory_extra.b2_00__4 =  'A4 for generic b2_00'
theory_extra.f1h_cBs =  'fraction multiplying 1-halo term, NcBs'
theory_extra.gamma_b1sB =  'the relative fraction of b1_sB to b1_cA'
theory_extra.b2_00__2 =  'A2 for generic b2_00'

#-------------------------------------------------------------------------------
# model params
#-------------------------------------------------------------------------------
model.Pdv_model_type = 'jennings'
model.correct_mu2 = False
model.correct_mu4 = False
model.params = 'runPB.ini'
model.fog_model = 'modified_lorentzian'
model.include_2loop = False
model.interpolate = True
model.max_mu = 4
model.transfer_fit = 'CLASS'
model.use_P00_model = True
model.use_P01_model = True
model.use_P11_model = True
model.use_Pdv_model = True
model.use_Phm_model = True
model.use_mean_bias = False
model.use_so_correction = False
model.use_tidal_bias = False
model.use_vlah_biasing = True
model.vel_disp_from_sims = False
model.z = 0.55
#-------------------------------------------------------------------------------
```

The parameter file uses a simple key/value assignment syntax to define the parameters. It is also possible to use environment variables when defining parameters, was was done in the above example for `$(PYRSD_DATA)`.

To learn more about each section of the parameter file, please see the next sections:

1. *Specifying the Data*
2. *Specifying the Theory*
3. *Running the Fits*

1.10 Specifying the Data

The user must specify the data measurements that he or she wishes to fit. Here, we provide an overview and examples of how to accomplish that using the `pyRSD.rsdfit` module.

1.10.1 Overview

The main class that is responsible for handling the data statistics, as well as the covariance matrix used during parameter estimation, is the *pyRSD.rsdfit.data.PowerData* class.

Information about the parameters that are needed to initialize the *PowerData* class can be found by using the *PowerData.help()* function,

```
In [1]: from pyRSD.rsdfit.data import PowerData

# print out the help message for the parameters needed
In [2]: PowerData.help()
Initialization Parameters for PowerData
-------------------------------------------------
covariance :

        The string specifying the name of the file holding the covariance matrix.

covariance_Nmocks :

        The number of mocks that was used to measure the covariance matrix.

        If this is non-zero, then the inverse covariance matrix will
        be rescaled to account for noise due to the finite number of mocks


        Default: 0.0

covariance_rescaling :

        Rescale the covariance matrix read from file by this amount.


        Default: 1.0

data_file :

        The string specifying the name of the file holding the data measurements.

ells :

        A list of integers specifying multipole numbers for each statistic
        in the final analysis.

        This must be supplied when the :attr:`mode` is ``poles``


        Default: None

fitting_range :

        The :math:`k` fitting range for each statistics.

        This can either be a tuple of (kmin, kmax), which will be
        used for each statistic or a list of tuples of (kmin, kmax)

grid_file :
```

(continues on next page)

(continued from previous page)

```
        A string specifying the name of the file holding a
        :class:`pyRSD.rsd.transfers.PkmuGrid` to read.


        Default: None

max_ellprime :

        When convolving a multipole of order ``ell``, include contributions
        up to and including this number.


        Default: 4

mode :

        The type of data, either ``pkmu`` or ``poles``

mu_bounds :

        A list of tuples specifying the edges of the :math:`\mu` bins.

        This should have (mu_min, mu_max), corresponding
        to the edges of the bins for each statistic in the final analysis

        This must be supplied when the :attr:`mode` is ``pkmu``


        Default: None

statistics :

        A list of the string names for each statistic that will be read from file

        These strings should be of the form:

        >> ['pole_0', 'pole_2', ...]
        >> ['pkmu_0.1', 'pkmu_0.3', ...]

usedata :

        A list of the statistic numbers that will be included in the final
        analysis.

        This allows the user to exclude certain statistics read from file. By
        default (``None``), all statistics are included


        Default: None

window_file :

        A string specifying the name of the file holding the correlation
        function multipoles of the window function.

        The file should contain columns of data, with the first column
        specifying the separation array :math:`s`, and the other columns
```

(continues on next page)

(continued from previous page)

```
        giving the even-numbered correlation function multipoles of the window


        Default: None

window_kmax :

        Default kmax value to use on the grid when convolving the model.


        Default: 0.7

window_kmin :

        Default kmin value to use on the grid when convolving the model.


        Default: 0.0001
```

These parameters should be specified in the parameter file that is passed to the `rsdfit` executable and the names of the parameters should be prefixed with the `data.` prefix. In our example parameter file discussed previously, we specify multipoles data to read from file as

```
#------------------------------------------------------------------------------
# data params
#------------------------------------------------------------------------------
data.covariance = '$(PYRSD_DATA)/examples/runPB_poles_gaussian_cov.dat'
data.covariance_Nmocks = 0
data.covariance_rescaling = 1.0
data.data_file = '$(PYRSD_DATA)/examples/runPB_galaxy_poles.dat'
data.ells = [0, 2, 4]
data.fitting_range = [(0.02, 0.4), (0.02, 0.4), (0.02, 0.4)]
data.grid_file = '$(PYRSD_DATA)/examples/runPB_pkmu_grid.dat'
data.mode = 'poles'
data.mu_bounds = None
data.statistics = ['pole_0', 'pole_2', 'pole_4']
data.usedata = range(0, 3)
data.window_file = None
#------------------------------------------------------------------------------
```

These parameters allow the user to specify which type of data is being used by specifying the `mode` parameter, either `pkmu` for $P(k, \mu)$ data or `poles` for $P_\ell(k)$ data. The user can also specify the desired k ranges to use when fitting the data, via the `fitting_range` parameter.

The data itself must be read in from a plaintext file. Similarly, the covariance matrix and grid file must also be read from a plaintext file. See the next section *File Formats for Data Files* for more specifics on the format of these plaintext files.

1.10.2 File Formats for Data Files

```
In [1]: import os

In [2]: startdir = os.path.abspath('.')

In [3]: home = startdir.rsplit('docs' , 1)[0]
```

(continues on next page)

(continued from previous page)

```
In [4]: os.chdir(home);

In [5]: os.chdir('docs/source')

In [6]: if not os.path.exists('generated'):
   ...:   os.makedirs('generated')
   ...:

In [7]: os.chdir('generated')
```

The data statistics, covariance matrix, and grid file are all read from plaintext files by the `pyRSD.rsdfit` module, and each requires a special format. We will describe those file file formats in this section.

The Power Statistics

The plaintext file holding the data statistics must be specified in the parameter file using the `data.data_file` parameter. The package can handle reading either $P(k,\mu)$ or $P_\ell(k)$ data, and we will give examples for both below.

$P(k,\mu)$ Data

For $P(k,\mu)$ data, we have power measured in wide bins of μ. If there are N_k k bins and N_μ μ bins, the pyRSD code expects the following data:

1. **k** :

   an array of shape (N_k, N_μ) providing the mean wavenumber value in each bin (units: h/Mpc)

2. **mu** :

   an array of shape (N_k, N_μ) providing the mean μ value in each bin

3. **power** :

   an array of shape (N_k, N_μ) providing the mean power value in each bin (units: $h^{-3}\mathrm{Mpc}^3$)

The data can be saved easily to a plaintext file from a numpy structured array using the *pyRSD.rsdfit.data.PowerMeasurements* class. For example, here we will generate some fake $P(k,\mu)$ and write this data to a plaintext file with the correct format.

```
In [8]: from pyRSD.rsdfit.data import PowerMeasurements

In [9]: import numpy as np

In [10]: Nk = 20

In [11]: Nmu = 5

# generate 5 mu bins from 0 to 1
In [12]: mu_edges = np.linspace(0, 1, Nmu+1)

In [13]: mu_cen = 0.5 * (mu_edges[1:] + mu_edges[:-1])

# generate 20 k bins from 0.01 to 0.4
In [14]: k_edges = np.linspace(0.01, 0.4, Nk+1)
```

(continues on next page)

(continued from previous page)

```
In [15]: k_cen = 0.5 * (k_edges[1:] + k_edges[:-1])

# make 2D k, mu arrays with shape (20, 5)
In [16]: k, mu = np.meshgrid(k_cen, mu_cen, indexing='ij')

# some random power data
In [17]: pkmu = np.random.random(size=(Nk,Nmu))

# make a structured array
In [18]: data = np.empty((Nk,Nmu), dtype=[('k', 'f8'), ('mu', 'f8'), ('power', 'f8')])

In [19]: data['k'] = k[:]

In [20]: data['mu'] = mu[:]

In [21]: data['power'] = pkmu[:]

# identifying names for each statistic
In [22]: names = ['pkmu_0.1', 'pkmu_0.3', 'pkmu_0.5', 'pkmu_0.7', 'pkmu_0.9']

# initialize the PowerMeasurements object
In [23]: measurements = PowerMeasurements.from_array(names, data)

In [24]: measurements
Out[24]:
[<PowerMeasurement P(k, mu=0.1), (0.0198 - 0.39) h/Mpc, 20 data points>,
 <PowerMeasurement P(k, mu=0.3), (0.0198 - 0.39) h/Mpc, 20 data points>,
 <PowerMeasurement P(k, mu=0.5), (0.0198 - 0.39) h/Mpc, 20 data points>,
 <PowerMeasurement P(k, mu=0.7), (0.0198 - 0.39) h/Mpc, 20 data points>,
 <PowerMeasurement P(k, mu=0.9), (0.0198 - 0.39) h/Mpc, 20 data points>]

# save to file
In [25]: measurements.to_plaintext("pkmu_data.dat")
```

Our fake data has been saved to a plaintext text file with the desired format. The first few lines of this plaintext file look like:

```
20 5
k mu power
1.97500e-02 1.00000e-01 7.64224e-01
3.92500e-02 1.00000e-01 8.56222e-01
5.87500e-02 1.00000e-01 6.61978e-01
7.82500e-02 1.00000e-01 6.40094e-01
9.77500e-02 1.00000e-01 6.17322e-01
1.17250e-01 1.00000e-01 6.70898e-01
1.36750e-01 1.00000e-01 8.24965e-01
1.56250e-01 1.00000e-01 7.94638e-01
```

We can easily re-initialize a `PowerMeasurements` object from this plaintext file using

```
In [26]: names = ['pkmu_0.1', 'pkmu_0.3', 'pkmu_0.5', 'pkmu_0.7', 'pkmu_0.9']

In [27]: measurements_2 = PowerMeasurements.from_plaintext(names, 'pkmu_data.dat')

In [28]: measurements_2
Out[28]:
```

(continues on next page)

(continued from previous page)

```
[<PowerMeasurement P(k, mu=0.1), (0.0198 - 0.39) h/Mpc, 20 data points>,
 <PowerMeasurement P(k, mu=0.3), (0.0198 - 0.39) h/Mpc, 20 data points>,
 <PowerMeasurement P(k, mu=0.5), (0.0198 - 0.39) h/Mpc, 20 data points>,
 <PowerMeasurement P(k, mu=0.7), (0.0198 - 0.39) h/Mpc, 20 data points>,
 <PowerMeasurement P(k, mu=0.9), (0.0198 - 0.39) h/Mpc, 20 data points>]
```

Note: The `names` specified in this example serve as identifying strings for each power bin. They should be specified via the `data.statistics` keyword in the parameter file. For $P(k,\mu)$ data, the names must begin with `pkmu_` and are typically followed by the mean μ value in each power bin, i.e., `pkmu_0.1` identifies the statistic measuring $P(k,\mu)$ in a bin centered at $\mu = 0.1$.

$P_\ell(k)$ Data

For $P_\ell(k)$ data, we have measured the multipoles moments of $P(k,\mu)$ for several multipole numbers ℓ. If there are N_k k bins and N_ℓ multipoles, the pyRSD code expects the following data:

1. **k** :

   an array of shape (N_k, N_ℓ) providing the mean wavenumber for each multipole (units: h/Mpc)

3. **power** :

   an array of shape (N_k, N_ℓ) providing the mean power for each multipole (units: $h^{-3}\mathrm{Mpc}^3$)

Again, here we will generate some fake $P_\ell(k)$ and write this data to a plaintext file with the correct format as an example.

```
In [29]: Nk = 20

In [30]: Nell = 3

# fit the monopole, quadrupole, and hexadecapole
In [31]: ells = [0, 2, 4]

# generate 20 k bins from 0.01 to 0.4
In [32]: k_edges = np.linspace(0.01, 0.4, Nk+1)

In [33]: k_cen = 0.5 * (k_edges[1:] + k_edges[:-1])

# make a structured array
In [34]: data = np.empty((Nk,Nell), dtype=[('k', 'f8'), ('power', 'f8')])

In [35]: for i, ell in enumerate(ells):
   ....:     data['k'][:,i] = k_cen[:]
   ....:     data['power'][:,i] = np.random.random(size=Nk)
   ....:

# identifying names for each statistic
In [36]: names = ['pole_0', 'pole_2', 'pole_4']

# initialize the PowerMeasurements object
In [37]: measurements = PowerMeasurements.from_array(names, data)

In [38]: measurements
```

(continues on next page)

(continued from previous page)

```
Out[38]:
[<PowerMeasurement P(k, ell=0), (0.0198 - 0.39) h/Mpc, 20 data points>,
 <PowerMeasurement P(k, ell=2), (0.0198 - 0.39) h/Mpc, 20 data points>,
 <PowerMeasurement P(k, ell=4), (0.0198 - 0.39) h/Mpc, 20 data points>]

# save to file
In [39]: measurements.to_plaintext("pole_data.dat")
```

Our fake data has been saved to a plaintext text file with the desired format. The first few lines of this plaintext file look like:

```
20 3
k power
1.97500e-02 4.19261e-02
3.92500e-02 4.82321e-01
5.87500e-02 8.89154e-01
7.82500e-02 9.52871e-01
9.77500e-02 6.73296e-01
1.17250e-01 3.76759e-02
1.36750e-01 5.56608e-01
1.56250e-01 4.39279e-02
```

We can easily re-initialize a `PowerMeasurements` object from this plaintext file using

```
In [40]: names = ['pole_0', 'pole_2', 'pole_4']

In [41]: measurements_2 = PowerMeasurements.from_plaintext(names, 'pole_data.dat')

In [42]: measurements_2
Out[42]:
[<PowerMeasurement P(k, ell=0), (0.0198 - 0.39) h/Mpc, 20 data points>,
 <PowerMeasurement P(k, ell=2), (0.0198 - 0.39) h/Mpc, 20 data points>,
 <PowerMeasurement P(k, ell=4), (0.0198 - 0.39) h/Mpc, 20 data points>]
```

Note: The `names` specified in this example serve as identifying strings for each multipole. They should be specified via the `data.statistics` keyword in the parameter file. For $P_\ell(k)$ data, the names must begin with `pole_` and are typically followed by the multipole number, i.e., `pole_0` identifies the monopole ($\ell = 0$).

The Covariance Matrix

The parameter estimation procedure relies on the likelihood function, which tells use the probability of the measured data given our theoretical model. In order to evaluate the likelihood, we need the covariance matrix of the data statistics. pyRSD provides two classes for dealing with covariance matrices, *pyRSD.rsdfit.data.PkmuCovarianceMatrix* for $P(k, \mu)$ data and *pyRSD.rsdfit.data.PoleCovarianceMatrix* for $P_\ell(k)$ data.

Again, when running the `rsdfit` command, the covariance matrix will be loaded from a plaintext file with a specific format. The easiest way to save your desired covariance matrix is take advantage of the functionality provided by the *PkmuCovarianceMatrix* and *pyRSD.rsdfit.data.PoleCovarianceMatrix* classes.

The most important thing to realize when dealing with the covariance matrix is that it is the covariance matrix of the full, concatenated data vector. For example, if we are fitting the $\ell = 0, 2, 4$ multipoles, then the full data vector is

$$\mathcal{D} = [P_0, P_2, P_4].$$

Then, if we have N_k data points measured for each multipole, then the covariance matrix has a size of $(3N_k, 3N_k)$. The upper left sub-matrix of size (N_k, N_k) is the covariance of P_0 with itself, the next sub-matrix of size (N_k, N_k) to the right is the covariance between P_0 and P_2 and similarly, the last sub-matrix on the top row is the covariance between P_0 and P_4.

As an example, we will generate some fake multipole data and compute the covariance below,

```
In [43]: from pyRSD.rsdfit.data import PoleCovarianceMatrix

# generate 100 fake monopoles
In [44]: P0 = np.random.random(size=(100, Nk)) # Nk = 20, see above

# 100 fake quadrupoles
In [45]: P2 = np.random.random(size=(100, Nk))

# 100 fake hexadecapoles
In [46]: P4 = np.random.random(size=(100, Nk))

# make the full data vector
In [47]: D = np.concatenate([P0, P2, P4], axis=-1) # shape is (100, 60)

# compute the covariance matrix
In [48]: cov = np.cov(D, rowvar=False) # shape is (20,20)

# initialize the PoleCovarianceMatrix
In [49]: ells = [0,2,4]

In [50]: C = PoleCovarianceMatrix(cov, k_cen, ells)

In [51]: C
Out[51]: <PoleCovarianceMatrix: size 60x60>

# write to plaintext file
In [52]: C.to_plaintext('pole_cov.dat')
```

The covariance matrix of our fake data has been saved to a plaintext text file with the desired format. The first few lines of this plaintext file look like:

```
60
0.08588355840145967
-0.02322592330872416
0.0034207681593465974
0.011111524674156586
0.008022754211141564
0.0016774454095546064
-0.012605623886929901
-0.0019657329320749025
0.0032883878524930083
```

We can easily re-initialize a `PoleCovarianceMatrix` object from this plaintext file using

```
In [53]: names = ['pole_0', 'pole_2', 'pole_4']

In [54]: C_2 = PoleCovarianceMatrix.from_plaintext('pole_cov.dat')

In [55]: C_2
Out[55]: <PoleCovarianceMatrix: size 60x60>
```

The case of $P(k, \mu)$ data is very similar to multipoles, except now the second dimension specifies the μ bins, rather

than the multipole numbers. The class `PkmuCovarianceMatrix` should be used for $P(k, \mu)$ data.

Warning: Be sure to ensure that the number of data points, specifically the k range used, of the data statistics in the file specified by the `data.data_file` attribute agrees with the number of elements in the covariance matrix.

The Window Function

If the user wishes to fit window-convolved power spectrum multipole, a file holding the correlation function multipoles of the window function should be specified using the `data.window_file` parameter. Then, the theoretical model will be convolved with this window function to accurately compare model and data.

The window function file should hold several columns of data, with the first column specifying the separation array s, and the other columns giving the even-numbered correlation function multipoles of the window. See *Window-convolved Power Spectra* for a full example of the window multipoles.

The $P(k, \mu)$ Grid

As discussed in *Discrete Binning Effects*, the user can optionally take into account the effects of dicretely binned data with a finely-binned $P(k, \mu)$ grid. The name of the file holding this grid should be specified in the parameter file using the `data.grid_file` attribute.

Given a 2D data array specifies this grid, the easiest way to save the grid to a plaintext file in the right format is to use the `PkmuGrid.to_plaintext()` function. See *Discrete Binning Effects* for a full example of how to do this.

Note: If the user is fitting window-convolved multipoles, the grid file does not need to be specified.

1.10.3 Generating Gaussian Covariance Matrices

In this section, we describe how users can use either a *GalaxySpectrum* or `QuasarSpectrum` model instance to compute the analytic Gaussian covariance matrix for either multipole or $P(k, \mu)$ wedges using the *PoleCovarianceMatrix* and *PkmuCovarianceMatrix* class objects.

Surveys with a varying $n(z)$

For surveys with a number density of objects that varies with redshift, we provide the *PoleCovarianceMatrix.cutsky_gaussian_covariance()* function to compute a Gaussian estimate of the covariance. There are a few key parameters for computing this estimate:

- **model** : the pyRSD model object, used to compute $P(k, \mu)$ in the covariance calculation
- **nbar** : a callable function that returns the $n(z)$ of the survey
- **fsky** : the sky fraction that survey covers
- **k** : the wavenumbers where the covariance is evaluated, in units of h/Mpc
- **ells** : the multipoles to compute the covariance of
- **zmin, zmax** : the redshift range of the survey
- **P0_FKP** : the value of P_0 to use in the FKP weights, given by $1/(1 + n(z)P_0)$

Below is an example of how to create a *PoleCovarianceMatrix* object holding the Gaussian covariance estimate for the BOSS DR12 data set.

```
import numpy
from scipy.interpolate import InterpolatedUnivariateSpline as spline
from pyRSD.rsd import GalaxySpectrum
from pyRSD.rsdfit.data import PoleCovarianceMatrix

# load n(z) from file and interpolate it
filename = 'nbar_DR12v5_CMASSLOWZ_North_om0p31_Pfkp10000.dat'
nbar = numpy.loadtxt(filename, skiprows=3)
nbar = spline(nbar[:,0], nbar[:,3])

# the sky fraction the survey covers
fsky = 0.1436

# the model instance to compute P(k,mu)
model = GalaxySpectrum(params='boss_dr12_fidcosmo.ini')

# the k values to compute covariance at
k = numpy.arange(0., 0.4, 0.005) + 0.005/2

# the multipoles to compute covariance of
ells = [0,2,4]

# the redshift range of the survey
zmin = 0.2
zmax = 0.5

# the FKP weight P0
P0_FKP = 1e4

# the PoleCovarianceMatrix holding the Gaussian covariance
C = PoleCovarianceMatrix.cutsky_gaussian_covariance(model, k, ells, nbar, fsky, zmin,␣
↪zmax, P0_FKP=P0_FKP)
```

Warning: If using the Gaussian covariance in parameter fits, be sure to ensure that the `k` parameter used in the covariance matrix calculation agrees with k range used for the data statistics in the file specified by the `data.data_file`.

Simulation boxes with a constant $\bar{n}$

For periodic simulation boxes with a constant number density $\bar{n}$, we can compute the Gaussian covariance matrix of multipoles using *PoleCovarianceMatrix.periodic_gaussian_covariance()* and for $P(k, \mu)$ wedges using *PkmuCovarianceMatrix.periodic_gaussian_covariance()*. These estimates can be computed by specifying the number density $\bar{n}$ and the volume of the simulation box. For more details, see equations 16 and 17 of Grieb et al. 2015.

For example, we can generate a *PkmuCovarianceMatrix* object holding the Gaussian wedge covariance using:

```
import numpy
from pyRSD.rsd import GalaxySpectrum
from pyRSD.rsdfit.data import PkmuCovarianceMatrix

# volume of the box
```

(continues on next page)

(continued from previous page)

```
volume = 1380.0**3

# constant number density in the box
nbar = 3e-4

# the RSD model
model = GalaxySpectrum(params='boss_dr12_fidcosmo.ini')

# evaluate the covariance at these wavenumbers
k = numpy.arange(0., 0.4, 0.005) + 0.005/2

# the edges of the P(k,mu) wedges
mu_edges = [0., 0.2, 0.4, 0.6, 0.8, 1.0]

# the PkmuCovarianceMatrix holding the Gaussian covariance
C = PkmuCovarianceMatrix.periodic_gaussian_covariance(model, k, ells, nbar, volume)
```

And, we can generate a *PoleCovarianceMatrix* object holding the Gaussian multipole covariance using:

```
import numpy
from pyRSD.rsd import GalaxySpectrum
from pyRSD.rsdfit.data import PoleCovarianceMatrix

# volume of the box
volume = 1380.0**3

# constant number density of the box
nbar = 3e-4

# model for computing P(k,mu)^2 in covariance calculation
model = GalaxySpectrum(params='boss_dr12_fidcosmo.ini')

# the wavenumbers where the covariance is evaluated
k = numpy.arange(0., 0.4, 0.005) + 0.005/2

# compute the covariance of these multipoles
ells = [0,2,4]

# the PoleCovarianceMatrix holding the Gaussian covariance
C = PoleCovarianceMatrix.periodic_gaussian_covariance(model, k, ells, nbar, volume)
```

Warning: If using the Gaussian covariance in parameter fits, be sure to ensure that the `k` parameter used in the covariance matrix calculation agrees with k range used for the data statistics in the file specified by the `data.data_file`.

1.10.4 API

The main class for handling the power measurements is the *PowerData* class. To initialize this class, the parameters are:

covariance	The string specifying the name of the file holding the covariance matrix.

Continued on next page

Table 12 – continued from previous page

covariance_Nmocks	The number of mocks that was used to measure the covariance matrix.
covariance_rescaling	Rescale the covariance matrix read from file by this amount.
data_file	The string specifying the name of the file holding the data measurements.
ells	A list of integers specifying multipole numbers for each statistic in the final analysis.
fitting_range	The k fitting range for each statistics.
grid_file	A string specifying the name of the file holding a `pyRSD.rsd.transfers.PkmuGrid` to read.
max_ellprime	When convolving a multipole of order `ell`, include contributions up to and including this number.
mode	The type of data, either `pkmu` or `poles`
mu_bounds	A list of tuples specifying the edges of the μ bins.
statistics	A list of the string names for each statistic that will be read from file
usedata	A list of the statistic numbers that will be included in the final analysis.
window_file	A string specifying the name of the file holding the correlation function multipoles of the window function.

class pyRSD.rsdfit.data.**PowerData**(*param_file*)

Class to hold several *PowerMeasurement* objects and combine the associated covariance matrices

Note: See the *PowerData.help()* function for a list of the parameters needed to initialize this class

Parameters **param_file** : str

the name of the parameter file holding the necessary parameters needed to initialize the object

covariance

The string specifying the name of the file holding the covariance matrix.

covariance_Nmocks

The number of mocks that was used to measure the covariance matrix.

If this is non-zero, then the inverse covariance matrix will be rescaled to account for noise due to the finite number of mocks

covariance_rescaling

Rescale the covariance matrix read from file by this amount.

data_file

The string specifying the name of the file holding the data measurements.

ells

A list of integers specifying multipole numbers for each statistic in the final analysis.

This must be supplied when the *mode* is `poles`

fitting_range

The k fitting range for each statistics.

This can either be a tuple of (kmin, kmax), which will be used for each statistic or a list of tuples of (kmin, kmax)

grid_file
A string specifying the name of the file holding a `pyRSD.rsd.transfers.PkmuGrid` to read.

static help()
Print out the help information for the necessary initialization parameters

max_ellprime
When convolving a multipole of order `ell`, include contributions up to and including this number.

mode
The type of data, either `pkmu` or `poles`

mu_bounds
A list of tuples specifying the edges of the μ bins.

This should have (mu_min, mu_max), corresponding to the edges of the bins for each statistic in the final analysis

This must be supplied when the *mode* is `pkmu`

statistics
A list of the string names for each statistic that will be read from file

These strings should be of the form:

>> ['pole_0', 'pole_2', ...] >> ['pkmu_0.1', 'pkmu_0.3', ...]

to_file(*filename*, *mode='w'*)
Save the parameters of this data class to a file

Parameters **filename** : str

the name of the file to write out the parameters to

mode : str

the mode to use when writing the parameters to file; i.e., 'w', 'a'

usedata
A list of the statistic numbers that will be included in the final analysis.

This allows the user to exclude certain statistics read from file. By default (`None`), all statistics are included

window_file
A string specifying the name of the file holding the correlation function multipoles of the window function.

The file should contain columns of data, with the first column specifying the separation array s, and the other columns giving the even-numbered correlation function multipoles of the window

Power Statistics

class pyRSD.rsdfit.data.**PowerMeasurements**
A list of `PowerMeasurement` objects

classmethod from_array(*names*, *data*)
Create a PowerMeasurement from the input structured array of data

Parameters **names** : list of str

the list of names for each measurement to load; each name must begin with `pkmu_` or `pole_` depending on the type of data. Examples of names include `pkmu_0.1` for a $\mu = 0.1$ $P(k, \mu)$ bin, or `pole_0` for the monopole, `pole_2` for the quadrupole, etc

data : array_like

a structured array holding the data fields; this have *k*, *mu*, *power* fields for $P(k, \mu)$ data or *k*, *power* fields for multipole data

classmethod from_plaintext(*names*, *filename*)

Load a set of power measurements from a plaintex file

Parameters **names** : list of str

the list of names for each measurement to load; each name must begin with `pkmu_` or `pole_` depending on the type of data. Examples of names include `pkmu_0.1` for a $\mu = 0.1$ $P(k, \mu)$ bin, or `pole_0` for the monopole, `pole_2` for the quadrupole, etc

filename : str

the name of the file to load to load a structured array of data from

to_plaintext(*filename*)

Write out the data from the power measurements to a plaintext file

Covariance Matrix

class pyRSD.rsdfit.data.**PoleCovarianceMatrix**(*data*, *k_center*, *ells*, ***kwargs*)

Class to hold a covariance matrix for multipole measurements

Parameters **data** : array_like (N,) or (N, N)

The data representing the covariance matrix. If 1D, the input is interpreted as the diagonal elements, with all other elements zero

k_center : array_like, (N,)

The wavenumbers where the center of the measurement k-bins are defined, for each element of the data matrix along one axis

ells : array_like, (N,)

The multipole numbers for each element of the data matrix along one axis

Examples

```
>>> from pyRSD.rsdfit.data import PoleCovarianceMatrix
>>> import numpy as np
```

```
>>> # generate 20 k bins from 0.01 to 0.4
>>> Nk = 20
>>> k_edges = np.linspace(0.01, 0.4, Nk+1)
>>> k_cen = 0.5 * (k_edges[1:] + k_edges[:-1])
```

```
>>> # generate 100 fake monopoles
>>> P0 = np.random.random(size=(100, Nk)) # Nk = 20, see above
```

```
>>> # 100 fake quadrupoles
>>> P2 = np.random.random(size=(100, Nk))
```

```
>>> # 100 fake hexadecapoles
>>> P4 = np.random.random(size=(100, Nk))
```

```
>>> # make the full data vector
>>> D = np.concatenate([P0, P2, P4], axis=-1) # shape is (100, 60)
```

```
>>> # compute the covariance matrix
>>> cov = np.cov(D, rowvar=False) # shape is (20,20)
```

```
>>> # initialize the PoleCovarianceMatrix
>>> ells = [0,2,4]
>>> C = PoleCovarianceMatrix(cov, k_cen, ells)
```

classmethod cutsky_gaussian_covariance(*model*, *k*, *ells*, *nbar*, *fsky*, *zmin*, *zmax*, *FKP_P0=10000.0*, *Nmu=100*, *Nz=50*)

Return the Gaussian prediction for the covariance between multipoles for a "cutsky" survey, i.e., a survey with a varying $n(z)$ distribution

Parameters **model** : GalaxySpectrum, QuasarSpectrum

the model instance used to evaluate the theoretical $P(k, \mu)$

k : array_like

the array of wavenumbers (units of h/Mpc) where the covariance matrix will be evaluated

ells : list of int

the list of multipole numbers to compute the covariance of

nbar : callable

a callable function returning the number density as a function of redshift

fsky : float

the sky fraction, used to normalized the effective volume calculation

zmin : float

the minimum redshift value to include when performing the effective volume calculation

zmax : float

the maximum redshift value to include when performing the effective volume calculation

FKP_P0 : float, optional

the FKP P0 value to use, where the FKP weights are $1/(1 + n(z)P_0)$; default is 1e4

Nmu : int, optional

the number of mu bins to use when performing the multipole integration over μ

Nz : int, optional

the number of redshift bins to use when performing the integral over redshift

Returns PoleCovarianceMatrix :

the covariance matrix object holding the Gaussian prediction for the covariance between the specified multipoles

Examples

```
>>> import numpy
>>> from scipy.interpolate import InterpolatedUnivariateSpline as spline
>>> from pyRSD.rsd import GalaxySpectrum
>>> from pyRSD.rsdfit.data import PoleCovarianceMatrix
>>> filename = 'nbar_DR12v5_CMASSLOWZ_North_om0p31_Pfkp10000.dat'
>>> nbar = numpy.loadtxt(filename, skiprows=3)
>>> nbar = spline(nbar[:,0], nbar[:,3])
>>> fsky = 0.1436
>>> model = GalaxySpectrum(params='boss_dr12_fidcosmo.ini')
>>> k = numpy.arange(0., 0.4, 0.005) + 0.005/2
>>> ells = [0,2,4]
>>> zmin = 0.2
>>> zmax = 0.5
>>> C = PoleCovarianceMatrix.cutsky_gaussian_covariance(model, k, ells, nbar,␣
↪fsky, zmin, zmax)
```

classmethod from_plaintext(*filename*)
Load the covariance matrix from a plain text file

classmethod periodic_gaussian_covariance(*model*, *k*, *ells*, *nbar*, *volume*, *Nmu=100*)
Return the Gaussian prediction for the covariance between multipoles for a periodic box simulation, where the number density is constant.

See eq. 16 of Grieb et al. 2015 arxiv:1509.04293

Parameters **model** : GalaxySpectrum, QuasarSpectrum

the model instance used to evaluate the theoretical $P(k, \mu)$

k : array_like

the array of wavenumbers (units of h/Mpc) where the covariance matrix will be evaluated

ells : list of int

the list of multipole numbers to compute the covariance of

nbar : float

the constant number density in the box in units of $(\mathrm{Mpc}/h)^{-3}$ – the shot noise contribution to the covariance is the inverse of this value

volume : float

the volume of the box, in units of $(\mathrm{Mpc}/h)^3$

Nmu : int, optional

the number of mu bins to use when performing the multipole integration over μ

Returns PoleCovarianceMatrix :

the covariance matrix object holding the Gaussian prediction for the covariance between the specified multipoles

Examples

```
>>> import numpy
>>> from pyRSD.rsd import GalaxySpectrum
>>> from pyRSD.rsdfit.data import PoleCovarianceMatrix
>>> volume = 1380.0**3
>>> nbar = 3e-4
>>> model = GalaxySpectrum(params='boss_dr12_fidcosmo.ini')
>>> k = numpy.arange(0., 0.4, 0.005) + 0.005/2
>>> ells = [0,2,4]
>>> C = PoleCovarianceMatrix.periodic_gaussian_covariance(model, k, ells,␣
↪nbar, volume)
```

to_plaintext (*filename*)
Save the covariance matrix to a plain text file

class pyRSD.rsdfit.data.**PkmuCovarianceMatrix** (*data*, *k_center*, *mu_center*, ***kwargs*)
Class to hold a covariance matrix for P(k, mu)

Parameters **data** : array_like (N,) or (N, N)

The data representing the covariance matrix. If 1D, the input is interpreted as the diagonal elements, with all other elements zero

k_center : array_like

The wavenumbers where the center of the measurement k-bins are defined, for each element of the data matrix along one axis

mu_center : array_like,

The values where the center of the measurement mu-bins are defined, for each element of the data matrix along one axis

classmethod from_plaintext (*filename*)
Load the covariance matrix from a plain text file

classmethod periodic_gaussian_covariance (*model*, *k*, *mu_edges*, *nbar*, *volume*, *Nmu=100*)
Return the Gaussian prediction for the covariance between $P(k, \mu)$ wedges for a periodic box simulation, where the number density is constant.

See eq. 17 of Grieb et al. 2015 arxiv:1509.04293

Parameters **model** : GalaxySpectrum, QuasarSpectrum

the model instance used to evaluate the theoretical $P(k, \mu)$

k : array_like

the array of wavenumbers (units of h/Mpc) where the covariance matrix will be evaluated

mu_edges : array_like

the edges of the μ bins

nbar : float

the constant number density in the box in units of $(\mathrm{Mpc}/h)^{-3}$ – the shot noise contribution to the covariance is the inverse of this value

volume : float

the volume of the box, in units of $(\mathrm{Mpc}/h)^3$

Nmu : int, optional

the number of mu bins to use when performing the multipole integration over μ

Returns PkmuCovarianceMatrix :

the covariance matrix object holding the Gaussian prediction for the covariance between the specified wedges

Examples

```
>>> import numpy
>>> from pyRSD.rsd import GalaxySpectrum
>>> from pyRSD.rsdfit.data import PkmuCovarianceMatrix
>>> volume = 1380.0**3
>>> nbar = 3e-4
>>> model = GalaxySpectrum(params='boss_dr12_fidcosmo.ini')
>>> k = numpy.arange(0., 0.4, 0.005) + 0.005/2
>>> mu_edges = [0., 0.2, 0.4, 0.6, 0.8, 1.0]
>>> C = PkmuCovarianceMatrix.periodic_gaussian_covariance(model, k, ells,␣
↪nbar, volume)
```

to_plaintext (*filename*)

Save the covariance matrix to a plain text file

1.11 Specifying the Theory

The user must specify the desired configuration of the theoretical model, including which parameters to vary, keep fixed, etc during the parameter estimation. Here, we provide an overview of this process and a few examples to get up and running quickly.

1.11.1 Overview

The default model parametrization is the one described in Section 4.4 of Hand et al. 2017. See this section for a detailed discussion of the free and constrained parameters, as well as the priors used during parameter fitting. There are 13 free parameters that are varied during the fitting procedure.

The default set of parameters can be easily loaded from a *GalaxySpectrum* object as

```
In [1]: from pyRSD.rsd import GalaxySpectrum

In [2]: model = GalaxySpectrum()

In [3]: params = model.default_params()

In [4]: print(params)
Parameters
__________
<Parameter 'alpha'              value=1              (constrained)       no prior>
<Parameter 'alpha_drag'         value=1              (fixed, fiducial)   no prior>
<Parameter 'alpha_par'          value=1              (free, fiducial)    ␣
↪Uniform(lower=0.8, upper=1.2)>
```

(continues on next page)

(continued from previous page)

```
<Parameter 'alpha_perp'          value=1               (free, fiducial)    ␣
↪Uniform(lower=0.8, upper=1.2)>
<Parameter 'b1'                  value=2.1243          (constrained)        no prior>
<Parameter 'b1_c'                value=1.9981          (constrained)        no prior>
<Parameter 'b1_cA'               value=1.9             (free, fiducial)    ␣
↪Uniform(lower=1.2, upper=2.5)>
<Parameter 'b1_cB'               value=3.0028          (constrained)        no prior>
<Parameter 'b1_s'                value=3.211           (constrained)        no prior>
<Parameter 'b1_sA'               value=2.755           (constrained)        no prior>
<Parameter 'b1_sB'               value=3.895           (constrained)        no prior>
<Parameter 'b1sigma8'            value=1.2958          (constrained)        no prior>
<Parameter 'epsilon'             value=0               (constrained)        no prior>
<Parameter 'f'                   value=0.78            (free, fiducial)    ␣
↪Uniform(lower=0.6, upper=1.0)>
<Parameter 'f1h_cBs'             value=1               (fixed, fiducial)   Normal(mu=1.
↪0, sigma=0.75)>
<Parameter 'f1h_sBsB'            value=4               (free, fiducial)    Normal(mu=4.
↪0, sigma=1.0)>
<Parameter 'F_AP'                value=1               (constrained)        no prior>
<Parameter 'f_so'                value=0.03            (fixed, fiducial)   Normal(mu=0.
↪04, sigma=0.02)>
<Parameter 'fcB'                 value=0.088988        (constrained)        no prior>
<Parameter 'fs'                  value=0.104           (free, fiducial)    ␣
↪Uniform(lower=0.0, upper=0.25)>
<Parameter 'fsB'                 value=0.4             (free, fiducial)    ␣
↪Uniform(lower=0.0, upper=1.0)>
<Parameter 'fsigma8'             value=0.4758          (constrained)        no prior>
<Parameter 'gamma_b1cB'          value=0.4             (fixed, fiducial)   Normal(mu=0.
↪4, sigma=0.2)>
<Parameter 'gamma_b1sA'          value=1.45            (free, fiducial)    Normal(mu=1.
↪45, sigma=0.3)>
<Parameter 'gamma_b1sB'          value=2.05            (free, fiducial)    Normal(mu=2.
↪05, sigma=0.3)>
<Parameter 'N'                   value=0               (fixed, fiducial)   ␣
↪Uniform(lower=-500.0, upper=500.0)>
<Parameter 'nbar'                value=0.0003117       (fixed, fiducial)    no prior>
<Parameter 'NcBs'                value=40237           (constrained)        no prior>
<Parameter 'Nsat_mult'           value=2.4             (free, fiducial)    Normal(mu=2.
↪4, sigma=0.2)>
<Parameter 'NsBsB'               value=1.2853e+05      (constrained)        no prior>
<Parameter 'sigma8_z'            value=0.61            (free, fiducial)    ␣
↪Uniform(lower=0.3, upper=0.9)>
<Parameter 'sigma_c'             value=1               (free, fiducial)    ␣
↪Uniform(lower=0.0, upper=3.0)>
<Parameter 'sigma_sA'            value=3.5             (free, fiducial)    ␣
↪Uniform(lower=2.0, upper=8.0)>
<Parameter 'sigma_sB'            value=5               (fixed, fiducial)   ␣
↪Uniform(lower=3.0, upper=10.0)>
<Parameter 'sigma_so'            value=4               (fixed, fiducial)   ␣
↪Uniform(lower=1.0, upper=7)>

Constraints
-----------
F_AP = alpha_par/alpha_perp
NcBs = f1h_cBs / (fcB*(1 - fs)*nbar)
NsBsB = f1h_sBsB / (fsB**2 * fs**2 * nbar) * (fcB*(1 - fs) - fs*(1-fsB))
alpha = (alpha_perp**2 * alpha_par)**(1./3)
```

(continues on next page)

(continued from previous page)

```
b1 = (1 - fs)*b1_c + fs*b1_s
b1_c = (1 - fcB)*b1_cA + fcB*b1_cB
b1_cB = (1-fsB)/(1+fsB*(1./Nsat_mult - 1)) * b1_sA +  (1 - (1-fsB)/(1+fsB*(1./Nsat_
↪mult - 1))) * b1_sB
b1_s = (1 - fsB)*b1_sA + fsB*b1_sB
b1_sA = gamma_b1sA*b1_cA
b1_sB = gamma_b1sB*b1_cA
b1sigma8 = b1*sigma8_z
epsilon = (alpha_perp/alpha_par)**(-1./3) - 1.0
fcB = fs / (1 - fs) * (1 + fsB*(1./Nsat_mult - 1))
fsigma8 = f*sigma8_z
```

The `params` variable here is a *pyRSD.rsdfit.parameters.ParameterSet* object, which is a dictionary of *pyRSD.rsdfit.parameters.Parameter* objects. The *Parameter* object not only stores the value of the parameter (as the `value` attribute), but also stores information about the prior and whether the parameter is freely varied or constrained. For example, the satellite fraction `fs` can be accessed as

```
In [5]: fsat = params['fs']

In [6]: print(fsat.value)
0.104

# freely varying
In [7]: print(fsat.vary)
\\\\\\True

In [8]: fsat.value = 0.12  # change the value to 0.12

# uniform prior assumed by default
In [9]: print(fsat.prior)
Uniform(lower=0.0, upper=0.25)
```

The Free Parameters

The 13 free parameters by default are:

```
In [10]: for par in params.free:
   ....:   print(par)
   ....:
<Parameter 'Nsat_mult'          value=2.4             (free, fiducial)    Normal(mu=2.
↪4, sigma=0.2)>
<Parameter 'alpha_par'          value=1               (free, fiducial)    ␣
↪Uniform(lower=0.8, upper=1.2)>
<Parameter 'alpha_perp'         value=1               (free, fiducial)    ␣
↪Uniform(lower=0.8, upper=1.2)>
<Parameter 'b1_cA'              value=1.9             (free, fiducial)    ␣
↪Uniform(lower=1.2, upper=2.5)>
<Parameter 'f'                  value=0.78            (free, fiducial)    ␣
↪Uniform(lower=0.6, upper=1.0)>
<Parameter 'f1h_sBsB'           value=4               (free, fiducial)    Normal(mu=4.
↪0, sigma=1.0)>
<Parameter 'fs'                 value=0.12            (free)              ␣
↪Uniform(lower=0.0, upper=0.25)>
<Parameter 'fsB'                value=0.4             (free, fiducial)    ␣
↪Uniform(lower=0.0, upper=1.0)>
```

(continues on next page)

(continued from previous page)

```
<Parameter 'gamma_b1sA'          value=1.45            (free, fiducial)    Normal(mu=1.
↪45, sigma=0.3)>
<Parameter 'gamma_b1sB'          value=2.05            (free, fiducial)    Normal(mu=2.
↪05, sigma=0.3)>
<Parameter 'sigma8_z'            value=0.61            (free, fiducial)    ␣
↪Uniform(lower=0.3, upper=0.9)>
<Parameter 'sigma_c'             value=1               (free, fiducial)    ␣
↪Uniform(lower=0.0, upper=3.0)>
<Parameter 'sigma_sA'            value=3.5             (free, fiducial)    ␣
↪Uniform(lower=2.0, upper=8.0)>
```

Parameter	Name	Description
$N_{s,\mathrm{mult}}$	Nsat_mult	The mean number of satellites in halos with >1 sat
$\alpha_{\parallel}$	alpha_par	The Alcock-Paczynski effect parameter for parallel to the line-of-sight
$\alpha_{\perp}$	alpha_perp	The Alcock-Paczynski effect parameter for perpendicular to the line-of-sight
b_{1,c_A}	b1_cA	The linear bias of type A centrals (no satellites in the same halo)
f	f	The growth rate at z: $f = d\mathrm{ln}D/d\mathrm{ln}a$
f_{1h,s_Bs_B}	f1h_sBsB	An order unity amplitude value multiplying the 1-halo term, `NsBsB`
f_s	f_s	The satellite fraction, which is (total number of satellites / total number of galaxies)
f_{s_B}	f_sB	The type B satellites fraction, which is (total number of type B satellites / total number of satellites)
$\gamma_{b_{1,s_A}}$	gamma_b1sA	The relative fraction of `b1_sA` to `b1_cA`
$\gamma_{b_{1,s_B}}$	gamma_b1sB	The relative fraction of `b1_sB` to `b1_cA`
$\sigma_8(z)$	sigma8_z	The mass variance at $r = 8\ \mathrm{Mpc}/h$ at z
σ_c	sigma_c	The centrals FoG velocity dispersion, in units of Mpc/h
σ_{s_A}	sigma_sA	The type A satellites FoG velocity dispersion, in units of Mpc/h

The Constrained Parameters

There are several constrained parameters, i.e., parameters whose values are solely determined by other parameters, in the default configuration. Note also that since these parameters are not freely varied, they do not require priors.

```
In [11]: for par in params.constrained:
   ....:     print(par)
   ....:
<Parameter 'F_AP'                value=1               (constrained)       no prior>
<Parameter 'NcBs'                value=34872           (constrained)       no prior>
<Parameter 'NsBsB'               value=1.114e+05       (constrained)       no prior>
<Parameter 'alpha'               value=1               (constrained)       no prior>
<Parameter 'b1'                  value=2.1588          (constrained)       no prior>
<Parameter 'b1_c'                value=2.0153          (constrained)       no prior>
<Parameter 'b1_cB'               value=3.0028          (constrained)       no prior>
<Parameter 'b1_s'                value=3.211           (constrained)       no prior>
<Parameter 'b1_sA'               value=2.755           (constrained)       no prior>
<Parameter 'b1_sB'               value=3.895           (constrained)       no prior>
<Parameter 'b1sigma8'            value=1.3169          (constrained)       no prior>
<Parameter 'epsilon'             value=0               (constrained)       no prior>
<Parameter 'fcB'                 value=0.10455         (constrained)       no prior>
<Parameter 'fsigma8'             value=0.4758          (constrained)       no prior>
```

Parameter	Name	Description
F_{AP}	F_AP	The AP parameter, given by $alpha_{\parallel}/\alpha_{\perp}$
$N_{c_B s}$	N_cBs	The amplitude of the constant 1-halo term between type B centrals, [units: $(\mathrm{Mpc}/h)^3$]
$N_{s_B s_B}$	N_sBsB	The amplitude of the constant 1-halo term between type B satellites, [units: $(\mathrm{Mpc}/h)^3$]
α	alpha	The isotropic AP dilation, given by $(\alpha_{\perp}^2\alpha_{\parallel})^{1/3}$
b_1	b1	The total galaxy linear bias
$b_{1,c}$	b1_c	The linear bias of the central sample
b_{1,c_B}	b1_cB	The linear bias of type B centrals (1 or more satellite(s) in the same halo)
$b_{1,s}$	b1_s	The linear bias of the satellite sample
b_{1,s_A}	b1_sA	The linear bias of the type A satellites sample
b_{1,s_B}	b1_sB	The linear bias of the type B satellites sample
$b_1\sigma_8$	b1sigma8	The value of $b_1(z) \times \sigma_8(z)$
ϵ	epsilon	The anisotropic AP warping, given by $(\alpha_{\perp}/\alpha_{\parallel})^{-1/3} - 1$
f_{c_B}	f_cB	The type B centrals fraction, which is (total number of type B centrals / total number of centrals)
$f\sigma_8$	fsigma8	The value of $f(z) \times \sigma_8(z)$
σ_{s_B}	sigma_sB	The type B satellites FoG velocity dispersion, in units of Mpc/h

The *ParameterSet* object handles constrained parameters automatically. For example, in our default configuration, we have the parameter `fsigma8`, which is the product of the growth rate `f` and the mass variance `sigma8_z`. We can change the value of either `f` or `sigma8_z` and the value of `fsigma8_z` will reflect those changes. For example,

```
In [12]: print(params['fsigma8'])
<Parameter 'fsigma8'            value=0.4758          (constrained)       no prior>

In [13]: print(params['f']*params['sigma8_z'])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\0.
↪4758

In [14]: params['f'].value = 0.75

In [15]: print(params['fsigma8'])
<Parameter 'fsigma8'            value=0.4575          (constrained)       no prior>

In [16]: print(params['f']*params['sigma8_z'])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\0.
↪4575
```

1.11.2 Writing to a Parameter File

```

```

The desired theoretical model parameterization must be written out to the parameter file that will be passed to the `rsdfit` executable. The easiest way to do this is to use the `to_file()` function of the default parameter object.

The recommended workflow to configure the theory is:

1. Generate the default parameter set via the `GalaxySpectrum.default_params()` function.

   2. Make the desired changes to the parametrization, i.e., changing priors, etc

   3. Write to an exisiting file using the `to_file()` function.

For example, assuming we have an existing parameter file entitled `params.dat`, we can write out our parameters as

```
In [1]: from pyRSD.rsd import GalaxySpectrum

# get the default parameters from an existing model
In [2]: model = GalaxySpectrum()

In [3]: params = model.default_params()

# make any desired changes
# ....
# write out to file
In [4]: params.to_file('params.dat', mode='a') # append to this file
```

This will write out to the file each `Parameter` in the parameter set as a dictionary. Now, the `params.dat` looks like

```
#------------------------------------------------------------------------------
# theory params
#------------------------------------------------------------------------------
theory.F_AP = {'name': 'F_AP', 'value': 1.0, 'vary': False, 'expr': 'alpha_par/alpha_
↪perp', 'analytic': False}
theory.N = {'name': 'N', 'value': 0.0, 'vary': False, 'fiducial': 0.0, 'prior_name':
↪'uniform', 'lower': -500.0, 'upper': 500.0, 'analytic': False}
theory.NcBs = {'name': 'NcBs', 'value': 40236.785434927464, 'vary': False, 'expr':
↪'f1h_cBs / (fcB*(1 - fs)*nbar)', 'fiducial': 45000.0, 'analytic': False}
theory.NsBsB = {'name': 'NsBsB', 'value': 128534.1756949072, 'vary': False, 'expr':
↪'f1h_sBsB / (fsB**2 * fs**2 * nbar) * (fcB*(1 - fs) - fs*(1-fsB))', 'fiducial':␣
↪94500.0, 'analytic': False}
theory.Nsat_mult = {'name': 'Nsat_mult', 'value': 2.4, 'vary': True, 'min': 2.0,
↪'fiducial': 2.4, 'prior_name': 'normal', 'mu': 2.4, 'sigma': 0.2, 'analytic': False}
theory.alpha = {'name': 'alpha', 'value': 1.0, 'vary': False, 'expr': '(alpha_perp**2␣
↪* alpha_par)**(1./3)', 'analytic': False}
theory.alpha_drag = {'name': 'alpha_drag', 'value': 1.0, 'vary': False, 'fiducial': 1.
↪0, 'analytic': False}
theory.alpha_par = {'name': 'alpha_par', 'value': 1.0, 'vary': True, 'fiducial': 1.0,
↪'prior_name': 'uniform', 'lower': 0.8, 'upper': 1.2, 'analytic': False}
theory.alpha_perp = {'name': 'alpha_perp', 'value': 1.0, 'vary': True, 'fiducial': 1.
↪0, 'prior_name': 'uniform', 'lower': 0.8, 'upper': 1.2, 'analytic': False}
theory.b1 = {'name': 'b1', 'value': 2.124276, 'vary': False, 'expr': '(1 - fs)*b1_c +␣
↪fs*b1_s', 'analytic': False}
theory.b1_c = {'name': 'b1_c', 'value': 1.9981383928571428, 'vary': False, 'expr':
↪'(1 - fcB)*b1_cA + fcB*b1_cB', 'analytic': False}
theory.b1_cA = {'name': 'b1_cA', 'value': 1.9, 'vary': True, 'fiducial': 1.9, 'prior_
↪name': 'uniform', 'lower': 1.2, 'upper': 2.5, 'analytic': False}
theory.b1_cB = {'name': 'b1_cB', 'value': 3.0028260869565218, 'vary': False, 'expr':
↪'(1-fsB)/(1+fsB*(1./Nsat_mult - 1)) * b1_sA +  (1 - (1-fsB)/(1+fsB*(1./Nsat_mult -␣
↪1))) * b1_sB', 'fiducial': 2.84, 'analytic': False}
theory.b1_s = {'name': 'b1_s', 'value': 3.2109999999999994, 'vary': False, 'expr':
↪'(1 - fsB)*b1_sA + fsB*b1_sB', 'analytic': False}
theory.b1_sA = {'name': 'b1_sA', 'value': 2.755, 'vary': False, 'expr': 'gamma_
↪b1sA*b1_cA', 'fiducial': 2.63, 'analytic': False}
theory.b1_sB = {'name': 'b1_sB', 'value': 3.8949999999999996, 'vary': False, 'expr':
↪'gamma_b1sB*b1_cA', 'fiducial': 3.62, 'analytic': False}
theory.b1sigma8 = {'name': 'b1sigma8', 'value': 1.29580836, 'vary': False, 'expr':
↪'b1*sigma8_z', 'analytic': False}
theory.epsilon = {'name': 'epsilon', 'value': 0.0, 'vary': False, 'expr': '(alpha_
↪perp/alpha_par)**(-1./3) - 1.0', 'analytic': False}
theory.f = {'name': 'f', 'value': 0.78, 'vary': True, 'fiducial': 0.78, 'prior_name':
↪'uniform', 'lower': 0.6, 'upper': 1.0, 'analytic': False}
```

(continues on next page)

(continued from previous page)

```
theory.f1h_cBs = {'name': 'f1h_cBs', 'value': 1.0, 'vary': False, 'min': 0, 'fiducial
↪': 1.0, 'prior_name': 'normal', 'mu': 1.0, 'sigma': 0.75, 'analytic': False}
theory.f1h_sBsB = {'name': 'f1h_sBsB', 'value': 4.0, 'vary': True, 'min': 0.0,
↪'fiducial': 4.0, 'prior_name': 'normal', 'mu': 4.0, 'sigma': 1.0, 'analytic': False}
theory.f_so = {'name': 'f_so', 'value': 0.03, 'vary': False, 'fiducial': 0.03, 'prior_
↪name': 'normal', 'mu': 0.04, 'sigma': 0.02, 'analytic': False}
theory.fcB = {'name': 'fcB', 'value': 0.08898809523809524, 'vary': False, 'min': 0,
↪'max': 1, 'expr': 'fs / (1 - fs) * (1 + fsB*(1./Nsat_mult - 1))', 'fiducial': 0.089,
↪ 'analytic': False}
theory.fs = {'name': 'fs', 'value': 0.104, 'vary': True, 'min': 0.0, 'max': 1.0,
↪'fiducial': 0.104, 'prior_name': 'uniform', 'lower': 0.0, 'upper': 0.25, 'analytic
↪': False}
theory.fsB = {'name': 'fsB', 'value': 0.4, 'vary': True, 'min': 0.0, 'max': 1,
↪'fiducial': 0.4, 'prior_name': 'uniform', 'lower': 0.0, 'upper': 1.0, 'analytic':␣
↪False}
theory.fsigma8 = {'name': 'fsigma8', 'value': 0.4758, 'vary': False, 'expr':
↪'f*sigma8_z', 'analytic': False}
theory.gamma_b1cB = {'name': 'gamma_b1cB', 'value': 0.4, 'vary': False, 'min': 0.0,
↪'max': 1.0, 'fiducial': 0.4, 'prior_name': 'normal', 'mu': 0.4, 'sigma': 0.2,
↪'analytic': False}
theory.gamma_b1sA = {'name': 'gamma_b1sA', 'value': 1.45, 'vary': True, 'min': 1.0,
↪'fiducial': 1.45, 'prior_name': 'normal', 'mu': 1.45, 'sigma': 0.3, 'analytic':␣
↪False}
theory.gamma_b1sB = {'name': 'gamma_b1sB', 'value': 2.05, 'vary': True, 'min': 1.0,
↪'fiducial': 2.05, 'prior_name': 'normal', 'mu': 2.05, 'sigma': 0.3, 'analytic':␣
↪False}
theory.nbar = {'name': 'nbar', 'value': 0.0003117, 'vary': False, 'fiducial': 0.
↪0003117, 'analytic': False}
theory.sigma8_z = {'name': 'sigma8_z', 'value': 0.61, 'vary': True, 'fiducial': 0.61,
↪'prior_name': 'uniform', 'lower': 0.3, 'upper': 0.9, 'analytic': False}
theory.sigma_c = {'name': 'sigma_c', 'value': 1.0, 'vary': True, 'fiducial': 1.0,
↪'prior_name': 'uniform', 'lower': 0.0, 'upper': 3.0, 'analytic': False}
theory.sigma_sA = {'name': 'sigma_sA', 'value': 3.5, 'vary': True, 'fiducial': 3.5,
↪'prior_name': 'uniform', 'lower': 2.0, 'upper': 8.0, 'analytic': False}
theory.sigma_sB = {'name': 'sigma_sB', 'value': 5.0, 'vary': False, 'fiducial': 5.0,
↪'prior_name': 'uniform', 'lower': 3.0, 'upper': 10.0, 'analytic': False}
theory.sigma_so = {'name': 'sigma_so', 'value': 4.0, 'vary': False, 'fiducial': 4.0,
↪'prior_name': 'uniform', 'lower': 1.0, 'upper': 7, 'analytic': False}
#------------------------------------------------------------------------------

#------------------------------------------------------------------------------
# theory params
#------------------------------------------------------------------------------
theory.F_AP = {'name': 'F_AP', 'value': 1.0, 'vary': False, 'expr': 'alpha_par/alpha_
↪perp', 'analytic': False}
theory.N = {'name': 'N', 'value': 0.0, 'vary': False, 'fiducial': 0.0, 'prior_name':
↪'uniform', 'lower': -500.0, 'upper': 500.0, 'analytic': False}
theory.NcBs = {'name': 'NcBs', 'value': 40236.785434927464, 'vary': False, 'expr':
↪'f1h_cBs / (fcB*(1 - fs)*nbar)', 'fiducial': 45000.0, 'analytic': False}
theory.NsBsB = {'name': 'NsBsB', 'value': 128534.1756949072, 'vary': False, 'expr':
↪'f1h_sBsB / (fsB**2 * fs**2 * nbar) * (fcB*(1 - fs) - fs*(1-fsB))', 'fiducial':␣
↪94500.0, 'analytic': False}
theory.Nsat_mult = {'name': 'Nsat_mult', 'value': 2.4, 'vary': True, 'min': 2.0,
↪'fiducial': 2.4, 'prior_name': 'normal', 'mu': 2.4, 'sigma': 0.2, 'analytic': False}
theory.alpha = {'name': 'alpha', 'value': 1.0, 'vary': False, 'expr': '(alpha_perp**2␣
↪* alpha_par)**(1./3)', 'analytic': False}
theory.alpha_drag = {'name': 'alpha_drag', 'value': 1.0, 'vary': False, 'fiducial': 1.
↪0, 'analytic': False}
```

(continues on next page)

(continued from previous page)

```
theory.alpha_par = {'name': 'alpha_par', 'value': 1.0, 'vary': True, 'fiducial': 1.0,
↪'prior_name': 'uniform', 'lower': 0.8, 'upper': 1.2, 'analytic': False}
theory.alpha_perp = {'name': 'alpha_perp', 'value': 1.0, 'vary': True, 'fiducial': 1.
↪0, 'prior_name': 'uniform', 'lower': 0.8, 'upper': 1.2, 'analytic': False}
theory.b1 = {'name': 'b1', 'value': 2.124276, 'vary': False, 'expr': '(1 - fs)*b1_c +␣
↪fs*b1_s', 'analytic': False}
theory.b1_c = {'name': 'b1_c', 'value': 1.9981383928571428, 'vary': False, 'expr':
↪'(1 - fcB)*b1_cA + fcB*b1_cB', 'analytic': False}
theory.b1_cA = {'name': 'b1_cA', 'value': 1.9, 'vary': True, 'fiducial': 1.9, 'prior_
↪name': 'uniform', 'lower': 1.2, 'upper': 2.5, 'analytic': False}
theory.b1_cB = {'name': 'b1_cB', 'value': 3.0028260869565218, 'vary': False, 'expr':
↪'(1-fsB)/(1+fsB*(1./Nsat_mult - 1)) * b1_sA +  (1 - (1-fsB)/(1+fsB*(1./Nsat_mult -␣
↪1))) * b1_sB', 'fiducial': 2.84, 'analytic': False}
theory.b1_s = {'name': 'b1_s', 'value': 3.2109999999999994, 'vary': False, 'expr':
↪'(1 - fsB)*b1_sA + fsB*b1_sB', 'analytic': False}
theory.b1_sA = {'name': 'b1_sA', 'value': 2.755, 'vary': False, 'expr': 'gamma_
↪b1sA*b1_cA', 'fiducial': 2.63, 'analytic': False}
theory.b1_sB = {'name': 'b1_sB', 'value': 3.8949999999999996, 'vary': False, 'expr':
↪'gamma_b1sB*b1_cA', 'fiducial': 3.62, 'analytic': False}
theory.b1sigma8 = {'name': 'b1sigma8', 'value': 1.29580836, 'vary': False, 'expr':
↪'b1*sigma8_z', 'analytic': False}
theory.epsilon = {'name': 'epsilon', 'value': 0.0, 'vary': False, 'expr': '(alpha_
↪perp/alpha_par)**(-1./3) - 1.0', 'analytic': False}
theory.f = {'name': 'f', 'value': 0.78, 'vary': True, 'fiducial': 0.78, 'prior_name':
↪'uniform', 'lower': 0.6, 'upper': 1.0, 'analytic': False}
theory.f1h_cBs = {'name': 'f1h_cBs', 'value': 1.0, 'vary': False, 'min': 0, 'fiducial
↪': 1.0, 'prior_name': 'normal', 'mu': 1.0, 'sigma': 0.75, 'analytic': False}
theory.f1h_sBsB = {'name': 'f1h_sBsB', 'value': 4.0, 'vary': True, 'min': 0.0,
↪'fiducial': 4.0, 'prior_name': 'normal', 'mu': 4.0, 'sigma': 1.0, 'analytic': False}
theory.f_so = {'name': 'f_so', 'value': 0.03, 'vary': False, 'fiducial': 0.03, 'prior_
↪name': 'normal', 'mu': 0.04, 'sigma': 0.02, 'analytic': False}
theory.fcB = {'name': 'fcB', 'value': 0.08898809523809524, 'vary': False, 'min': 0,
↪'max': 1, 'expr': 'fs / (1 - fs) * (1 + fsB*(1./Nsat_mult - 1))', 'fiducial': 0.089,
↪ 'analytic': False}
theory.fs = {'name': 'fs', 'value': 0.104, 'vary': True, 'min': 0.0, 'max': 1.0,
↪'fiducial': 0.104, 'prior_name': 'uniform', 'lower': 0.0, 'upper': 0.25, 'analytic
↪': False}
theory.fsB = {'name': 'fsB', 'value': 0.4, 'vary': True, 'min': 0.0, 'max': 1,
↪'fiducial': 0.4, 'prior_name': 'uniform', 'lower': 0.0, 'upper': 1.0, 'analytic':␣
↪False}
theory.fsigma8 = {'name': 'fsigma8', 'value': 0.4758, 'vary': False, 'expr':
↪'f*sigma8_z', 'analytic': False}
theory.gamma_b1cB = {'name': 'gamma_b1cB', 'value': 0.4, 'vary': False, 'min': 0.0,
↪'max': 1.0, 'fiducial': 0.4, 'prior_name': 'normal', 'mu': 0.4, 'sigma': 0.2,
↪'analytic': False}
theory.gamma_b1sA = {'name': 'gamma_b1sA', 'value': 1.45, 'vary': True, 'min': 1.0,
↪'fiducial': 1.45, 'prior_name': 'normal', 'mu': 1.45, 'sigma': 0.3, 'analytic':␣
↪False}
theory.gamma_b1sB = {'name': 'gamma_b1sB', 'value': 2.05, 'vary': True, 'min': 1.0,
↪'fiducial': 2.05, 'prior_name': 'normal', 'mu': 2.05, 'sigma': 0.3, 'analytic':␣
↪False}
theory.nbar = {'name': 'nbar', 'value': 0.0003117, 'vary': False, 'fiducial': 0.
↪0003117, 'analytic': False}
theory.sigma8_z = {'name': 'sigma8_z', 'value': 0.61, 'vary': True, 'fiducial': 0.61,
↪'prior_name': 'uniform', 'lower': 0.3, 'upper': 0.9, 'analytic': False}
theory.sigma_c = {'name': 'sigma_c', 'value': 1.0, 'vary': True, 'fiducial': 1.0,
↪'prior_name': 'uniform', 'lower': 0.0, 'upper': 3.0, 'analytic': False}
```

(continues on next page)

(continued from previous page)

```
theory.sigma_sA = {'name': 'sigma_sA', 'value': 3.5, 'vary': True, 'fiducial': 3.5,
↪'prior_name': 'uniform', 'lower': 2.0, 'upper': 8.0, 'analytic': False}
theory.sigma_sB = {'name': 'sigma_sB', 'value': 5.0, 'vary': False, 'fiducial': 5.0,
↪'prior_name': 'uniform', 'lower': 3.0, 'upper': 10.0, 'analytic': False}
theory.sigma_so = {'name': 'sigma_so', 'value': 4.0, 'vary': False, 'fiducial': 4.0,
↪'prior_name': 'uniform', 'lower': 1.0, 'upper': 7, 'analytic': False}
#-------------------------------------------------------------------------------

#-------------------------------------------------------------------------------
# theory params
#-------------------------------------------------------------------------------
theory.F_AP = {'name': 'F_AP', 'value': 1.0, 'vary': False, 'expr': 'alpha_par/alpha_
↪perp', 'analytic': False}
theory.N = {'name': 'N', 'value': 0.0, 'vary': False, 'fiducial': 0.0, 'prior_name':
↪'uniform', 'lower': -500.0, 'upper': 500.0, 'analytic': False}
theory.NcBs = {'name': 'NcBs', 'value': 40236.785434927464, 'vary': False, 'expr':
↪'f1h_cBs / (fcB*(1 - fs)*nbar)', 'fiducial': 45000.0, 'analytic': False}
theory.NsBsB = {'name': 'NsBsB', 'value': 128534.1756949072, 'vary': False, 'expr':
↪'f1h_sBsB / (fsB**2 * fs**2 * nbar) * (fcB*(1 - fs) - fs*(1-fsB))', 'fiducial':␣
↪94500.0, 'analytic': False}
theory.Nsat_mult = {'name': 'Nsat_mult', 'value': 2.4, 'vary': True, 'min': 2.0,
↪'fiducial': 2.4, 'prior_name': 'normal', 'mu': 2.4, 'sigma': 0.2, 'analytic': False}
theory.alpha = {'name': 'alpha', 'value': 1.0, 'vary': False, 'expr': '(alpha_perp**2␣
↪* alpha_par)**(1./3)', 'analytic': False}
theory.alpha_drag = {'name': 'alpha_drag', 'value': 1.0, 'vary': False, 'fiducial': 1.
↪0, 'analytic': False}
theory.alpha_par = {'name': 'alpha_par', 'value': 1.0, 'vary': True, 'fiducial': 1.0,
↪'prior_name': 'uniform', 'lower': 0.8, 'upper': 1.2, 'analytic': False}
theory.alpha_perp = {'name': 'alpha_perp', 'value': 1.0, 'vary': True, 'fiducial': 1.
↪0, 'prior_name': 'uniform', 'lower': 0.8, 'upper': 1.2, 'analytic': False}
theory.b1 = {'name': 'b1', 'value': 2.124276, 'vary': False, 'expr': '(1 - fs)*b1_c +␣
↪fs*b1_s', 'analytic': False}
theory.b1_c = {'name': 'b1_c', 'value': 1.9981383928571428, 'vary': False, 'expr':
↪'(1 - fcB)*b1_cA + fcB*b1_cB', 'analytic': False}
theory.b1_cA = {'name': 'b1_cA', 'value': 1.9, 'vary': True, 'fiducial': 1.9, 'prior_
↪name': 'uniform', 'lower': 1.2, 'upper': 2.5, 'analytic': False}
theory.b1_cB = {'name': 'b1_cB', 'value': 3.0028260869565218, 'vary': False, 'expr':
↪'(1-fsB)/(1+fsB*(1./Nsat_mult - 1)) * b1_sA +  (1 - (1-fsB)/(1+fsB*(1./Nsat_mult -␣
↪1))) * b1_sB', 'fiducial': 2.84, 'analytic': False}
theory.b1_s = {'name': 'b1_s', 'value': 3.2109999999999994, 'vary': False, 'expr':
↪'(1 - fsB)*b1_sA + fsB*b1_sB', 'analytic': False}
theory.b1_sA = {'name': 'b1_sA', 'value': 2.755, 'vary': False, 'expr': 'gamma_
↪b1sA*b1_cA', 'fiducial': 2.63, 'analytic': False}
theory.b1_sB = {'name': 'b1_sB', 'value': 3.8949999999999996, 'vary': False, 'expr':
↪'gamma_b1sB*b1_cA', 'fiducial': 3.62, 'analytic': False}
theory.b1sigma8 = {'name': 'b1sigma8', 'value': 1.29580836, 'vary': False, 'expr':
↪'b1*sigma8_z', 'analytic': False}
theory.epsilon = {'name': 'epsilon', 'value': 0.0, 'vary': False, 'expr': '(alpha_
↪perp/alpha_par)**(-1./3) - 1.0', 'analytic': False}
theory.f = {'name': 'f', 'value': 0.78, 'vary': True, 'fiducial': 0.78, 'prior_name':
↪'uniform', 'lower': 0.6, 'upper': 1.0, 'analytic': False}
theory.f1h_cBs = {'name': 'f1h_cBs', 'value': 1.0, 'vary': False, 'min': 0, 'fiducial
↪': 1.0, 'prior_name': 'normal', 'mu': 1.0, 'sigma': 0.75, 'analytic': False}
theory.f1h_sBsB = {'name': 'f1h_sBsB', 'value': 4.0, 'vary': True, 'min': 0.0,
↪'fiducial': 4.0, 'prior_name': 'normal', 'mu': 4.0, 'sigma': 1.0, 'analytic': False}
theory.f_so = {'name': 'f_so', 'value': 0.03, 'vary': False, 'fiducial': 0.03, 'prior_
↪name': 'normal', 'mu': 0.04, 'sigma': 0.02, 'analytic': False}
```

(continues on next page)

(continued from previous page)

```
theory.fcB = {'name': 'fcB', 'value': 0.08898809523809524, 'vary': False, 'min': 0,
↪'max': 1, 'expr': 'fs / (1 - fs) * (1 + fsB*(1./Nsat_mult - 1))', 'fiducial': 0.089,
↪ 'analytic': False}
theory.fs = {'name': 'fs', 'value': 0.104, 'vary': True, 'min': 0.0, 'max': 1.0,
↪'fiducial': 0.104, 'prior_name': 'uniform', 'lower': 0.0, 'upper': 0.25, 'analytic
↪': False}
theory.fsB = {'name': 'fsB', 'value': 0.4, 'vary': True, 'min': 0.0, 'max': 1,
↪'fiducial': 0.4, 'prior_name': 'uniform', 'lower': 0.0, 'upper': 1.0, 'analytic':␣
↪False}
theory.fsigma8 = {'name': 'fsigma8', 'value': 0.4758, 'vary': False, 'expr':
↪'f*sigma8_z', 'analytic': False}
theory.gamma_b1cB = {'name': 'gamma_b1cB', 'value': 0.4, 'vary': False, 'min': 0.0,
↪'max': 1.0, 'fiducial': 0.4, 'prior_name': 'normal', 'mu': 0.4, 'sigma': 0.2,
↪'analytic': False}
theory.gamma_b1sA = {'name': 'gamma_b1sA', 'value': 1.45, 'vary': True, 'min': 1.0,
↪'fiducial': 1.45, 'prior_name': 'normal', 'mu': 1.45, 'sigma': 0.3, 'analytic':␣
↪False}
theory.gamma_b1sB = {'name': 'gamma_b1sB', 'value': 2.05, 'vary': True, 'min': 1.0,
↪'fiducial': 2.05, 'prior_name': 'normal', 'mu': 2.05, 'sigma': 0.3, 'analytic':␣
↪False}
theory.nbar = {'name': 'nbar', 'value': 0.0003117, 'vary': False, 'fiducial': 0.
↪0003117, 'analytic': False}
theory.sigma8_z = {'name': 'sigma8_z', 'value': 0.61, 'vary': True, 'fiducial': 0.61,
↪'prior_name': 'uniform', 'lower': 0.3, 'upper': 0.9, 'analytic': False}
theory.sigma_c = {'name': 'sigma_c', 'value': 1.0, 'vary': True, 'fiducial': 1.0,
↪'prior_name': 'uniform', 'lower': 0.0, 'upper': 3.0, 'analytic': False}
theory.sigma_sA = {'name': 'sigma_sA', 'value': 3.5, 'vary': True, 'fiducial': 3.5,
↪'prior_name': 'uniform', 'lower': 2.0, 'upper': 8.0, 'analytic': False}
theory.sigma_sB = {'name': 'sigma_sB', 'value': 5.0, 'vary': False, 'fiducial': 5.0,
↪'prior_name': 'uniform', 'lower': 3.0, 'upper': 10.0, 'analytic': False}
theory.sigma_so = {'name': 'sigma_so', 'value': 4.0, 'vary': False, 'fiducial': 4.0,
↪'prior_name': 'uniform', 'lower': 1.0, 'upper': 7, 'analytic': False}
#-------------------------------------------------------------------------------

#-------------------------------------------------------------------------------
# theory params
#-------------------------------------------------------------------------------
theory.F_AP = {'name': 'F_AP', 'value': 1.0, 'vary': False, 'expr': 'alpha_par/alpha_
↪perp', 'analytic': False}
theory.N = {'name': 'N', 'value': 0.0, 'vary': False, 'fiducial': 0.0, 'prior_name':
↪'uniform', 'lower': -500.0, 'upper': 500.0, 'analytic': False}
theory.NcBs = {'name': 'NcBs', 'value': 40236.785434927464, 'vary': False, 'expr':
↪'f1h_cBs / (fcB*(1 - fs)*nbar)', 'fiducial': 45000.0, 'analytic': False}
theory.NsBsB = {'name': 'NsBsB', 'value': 128534.1756949072, 'vary': False, 'expr':
↪'f1h_sBsB / (fsB**2 * fs**2 * nbar) * (fcB*(1 - fs) - fs*(1-fsB))', 'fiducial':␣
↪94500.0, 'analytic': False}
theory.Nsat_mult = {'name': 'Nsat_mult', 'value': 2.4, 'vary': True, 'min': 2.0,
↪'fiducial': 2.4, 'prior_name': 'normal', 'mu': 2.4, 'sigma': 0.2, 'analytic': False}
theory.alpha = {'name': 'alpha', 'value': 1.0, 'vary': False, 'expr': '(alpha_perp**2␣
↪* alpha_par)**(1./3)', 'analytic': False}
theory.alpha_drag = {'name': 'alpha_drag', 'value': 1.0, 'vary': False, 'fiducial': 1.
↪0, 'analytic': False}
theory.alpha_par = {'name': 'alpha_par', 'value': 1.0, 'vary': True, 'fiducial': 1.0,
↪'prior_name': 'uniform', 'lower': 0.8, 'upper': 1.2, 'analytic': False}
theory.alpha_perp = {'name': 'alpha_perp', 'value': 1.0, 'vary': True, 'fiducial': 1.
↪0, 'prior_name': 'uniform', 'lower': 0.8, 'upper': 1.2, 'analytic': False}
theory.b1 = {'name': 'b1', 'value': 2.124276, 'vary': False, 'expr': '(1 - fs)*b1_c +␣
↪fs*b1_s', 'analytic': False}
```

(continues on next page)

(continued from previous page)

```
theory.b1_c = {'name': 'b1_c', 'value': 1.9981383928571428, 'vary': False, 'expr':
↪'(1 - fcB)*b1_cA + fcB*b1_cB', 'analytic': False}
theory.b1_cA = {'name': 'b1_cA', 'value': 1.9, 'vary': True, 'fiducial': 1.9, 'prior_
↪name': 'uniform', 'lower': 1.2, 'upper': 2.5, 'analytic': False}
theory.b1_cB = {'name': 'b1_cB', 'value': 3.0028260869565218, 'vary': False, 'expr':
↪'(1-fsB)/(1+fsB*(1./Nsat_mult - 1)) * b1_sA +  (1 - (1-fsB)/(1+fsB*(1./Nsat_mult -␣
↪1))) * b1_sB', 'fiducial': 2.84, 'analytic': False}
theory.b1_s = {'name': 'b1_s', 'value': 3.2109999999999994, 'vary': False, 'expr':
↪'(1 - fsB)*b1_sA + fsB*b1_sB', 'analytic': False}
theory.b1_sA = {'name': 'b1_sA', 'value': 2.755, 'vary': False, 'expr': 'gamma_
↪b1sA*b1_cA', 'fiducial': 2.63, 'analytic': False}
theory.b1_sB = {'name': 'b1_sB', 'value': 3.8949999999999996, 'vary': False, 'expr':
↪'gamma_b1sB*b1_cA', 'fiducial': 3.62, 'analytic': False}
theory.b1sigma8 = {'name': 'b1sigma8', 'value': 1.29580836, 'vary': False, 'expr':
↪'b1*sigma8_z', 'analytic': False}
theory.epsilon = {'name': 'epsilon', 'value': 0.0, 'vary': False, 'expr': '(alpha_
↪perp/alpha_par)**(-1./3) - 1.0', 'analytic': False}
theory.f = {'name': 'f', 'value': 0.78, 'vary': True, 'fiducial': 0.78, 'prior_name':
↪'uniform', 'lower': 0.6, 'upper': 1.0, 'analytic': False}
theory.f1h_cBs = {'name': 'f1h_cBs', 'value': 1.0, 'vary': False, 'min': 0, 'fiducial
↪': 1.0, 'prior_name': 'normal', 'mu': 1.0, 'sigma': 0.75, 'analytic': False}
theory.f1h_sBsB = {'name': 'f1h_sBsB', 'value': 4.0, 'vary': True, 'min': 0.0,
↪'fiducial': 4.0, 'prior_name': 'normal', 'mu': 4.0, 'sigma': 1.0, 'analytic': False}
theory.f_so = {'name': 'f_so', 'value': 0.03, 'vary': False, 'fiducial': 0.03, 'prior_
↪name': 'normal', 'mu': 0.04, 'sigma': 0.02, 'analytic': False}
theory.fcB = {'name': 'fcB', 'value': 0.08898809523809524, 'vary': False, 'min': 0,
↪'max': 1, 'expr': 'fs / (1 - fs) * (1 + fsB*(1./Nsat_mult - 1))', 'fiducial': 0.089,
↪ 'analytic': False}
theory.fs = {'name': 'fs', 'value': 0.104, 'vary': True, 'min': 0.0, 'max': 1.0,
↪'fiducial': 0.104, 'prior_name': 'uniform', 'lower': 0.0, 'upper': 0.25, 'analytic
↪': False}
theory.fsB = {'name': 'fsB', 'value': 0.4, 'vary': True, 'min': 0.0, 'max': 1,
↪'fiducial': 0.4, 'prior_name': 'uniform', 'lower': 0.0, 'upper': 1.0, 'analytic':␣
↪False}
theory.fsigma8 = {'name': 'fsigma8', 'value': 0.4758, 'vary': False, 'expr':
↪'f*sigma8_z', 'analytic': False}
theory.gamma_b1cB = {'name': 'gamma_b1cB', 'value': 0.4, 'vary': False, 'min': 0.0,
↪'max': 1.0, 'fiducial': 0.4, 'prior_name': 'normal', 'mu': 0.4, 'sigma': 0.2,
↪'analytic': False}
theory.gamma_b1sA = {'name': 'gamma_b1sA', 'value': 1.45, 'vary': True, 'min': 1.0,
↪'fiducial': 1.45, 'prior_name': 'normal', 'mu': 1.45, 'sigma': 0.3, 'analytic':␣
↪False}
theory.gamma_b1sB = {'name': 'gamma_b1sB', 'value': 2.05, 'vary': True, 'min': 1.0,
↪'fiducial': 2.05, 'prior_name': 'normal', 'mu': 2.05, 'sigma': 0.3, 'analytic':␣
↪False}
theory.nbar = {'name': 'nbar', 'value': 0.0003117, 'vary': False, 'fiducial': 0.
↪0003117, 'analytic': False}
theory.sigma8_z = {'name': 'sigma8_z', 'value': 0.61, 'vary': True, 'fiducial': 0.61,
↪'prior_name': 'uniform', 'lower': 0.3, 'upper': 0.9, 'analytic': False}
theory.sigma_c = {'name': 'sigma_c', 'value': 1.0, 'vary': True, 'fiducial': 1.0,
↪'prior_name': 'uniform', 'lower': 0.0, 'upper': 3.0, 'analytic': False}
theory.sigma_sA = {'name': 'sigma_sA', 'value': 3.5, 'vary': True, 'fiducial': 3.5,
↪'prior_name': 'uniform', 'lower': 2.0, 'upper': 8.0, 'analytic': False}
theory.sigma_sB = {'name': 'sigma_sB', 'value': 5.0, 'vary': False, 'fiducial': 5.0,
↪'prior_name': 'uniform', 'lower': 3.0, 'upper': 10.0, 'analytic': False}
theory.sigma_so = {'name': 'sigma_so', 'value': 4.0, 'vary': False, 'fiducial': 4.0,
↪'prior_name': 'uniform', 'lower': 1.0, 'upper': 7, 'analytic': False}
```

(continues on next page)

(continued from previous page)

```
#-------------------------------------------------------------------------------

#-------------------------------------------------------------------------------
# theory params
#-------------------------------------------------------------------------------
theory.F_AP = {'name': 'F_AP', 'value': 1.0, 'vary': False, 'expr': 'alpha_par/alpha_
↪perp', 'analytic': False}
theory.N = {'name': 'N', 'value': 0.0, 'vary': False, 'fiducial': 0.0, 'prior_name':
↪'uniform', 'lower': -500.0, 'upper': 500.0, 'analytic': False}
theory.NcBs = {'name': 'NcBs', 'value': 40236.785434927464, 'vary': False, 'expr':
↪'f1h_cBs / (fcB*(1 - fs)*nbar)', 'fiducial': 45000.0, 'analytic': False}
theory.NsBsB = {'name': 'NsBsB', 'value': 128534.1756949072, 'vary': False, 'expr':
↪'f1h_sBsB / (fsB**2 * fs**2 * nbar) * (fcB*(1 - fs) - fs*(1-fsB))', 'fiducial':␣
↪94500.0, 'analytic': False}
theory.Nsat_mult = {'name': 'Nsat_mult', 'value': 2.4, 'vary': True, 'min': 2.0,
↪'fiducial': 2.4, 'prior_name': 'normal', 'mu': 2.4, 'sigma': 0.2, 'analytic': False}
theory.alpha = {'name': 'alpha', 'value': 1.0, 'vary': False, 'expr': '(alpha_perp**2␣
↪* alpha_par)**(1./3)', 'analytic': False}
theory.alpha_drag = {'name': 'alpha_drag', 'value': 1.0, 'vary': False, 'fiducial': 1.
↪0, 'analytic': False}
theory.alpha_par = {'name': 'alpha_par', 'value': 1.0, 'vary': True, 'fiducial': 1.0,
↪'prior_name': 'uniform', 'lower': 0.8, 'upper': 1.2, 'analytic': False}
theory.alpha_perp = {'name': 'alpha_perp', 'value': 1.0, 'vary': True, 'fiducial': 1.
↪0, 'prior_name': 'uniform', 'lower': 0.8, 'upper': 1.2, 'analytic': False}
theory.b1 = {'name': 'b1', 'value': 2.124276, 'vary': False, 'expr': '(1 - fs)*b1_c +␣
↪fs*b1_s', 'analytic': False}
theory.b1_c = {'name': 'b1_c', 'value': 1.9981383928571428, 'vary': False, 'expr':
↪'(1 - fcB)*b1_cA + fcB*b1_cB', 'analytic': False}
theory.b1_cA = {'name': 'b1_cA', 'value': 1.9, 'vary': True, 'fiducial': 1.9, 'prior_
↪name': 'uniform', 'lower': 1.2, 'upper': 2.5, 'analytic': False}
theory.b1_cB = {'name': 'b1_cB', 'value': 3.0028260869565218, 'vary': False, 'expr':
↪'(1-fsB)/(1+fsB*(1./Nsat_mult - 1)) * b1_sA +  (1 - (1-fsB)/(1+fsB*(1./Nsat_mult -␣
↪1))) * b1_sB', 'fiducial': 2.84, 'analytic': False}
theory.b1_s = {'name': 'b1_s', 'value': 3.2109999999999994, 'vary': False, 'expr':
↪'(1 - fsB)*b1_sA + fsB*b1_sB', 'analytic': False}
theory.b1_sA = {'name': 'b1_sA', 'value': 2.755, 'vary': False, 'expr': 'gamma_
↪b1sA*b1_cA', 'fiducial': 2.63, 'analytic': False}
theory.b1_sB = {'name': 'b1_sB', 'value': 3.8949999999999996, 'vary': False, 'expr':
↪'gamma_b1sB*b1_cA', 'fiducial': 3.62, 'analytic': False}
theory.b1sigma8 = {'name': 'b1sigma8', 'value': 1.29580836, 'vary': False, 'expr':
↪'b1*sigma8_z', 'analytic': False}
theory.epsilon = {'name': 'epsilon', 'value': 0.0, 'vary': False, 'expr': '(alpha_
↪perp/alpha_par)**(-1./3) - 1.0', 'analytic': False}
theory.f = {'name': 'f', 'value': 0.78, 'vary': True, 'fiducial': 0.78, 'prior_name':
↪'uniform', 'lower': 0.6, 'upper': 1.0, 'analytic': False}
theory.f1h_cBs = {'name': 'f1h_cBs', 'value': 1.0, 'vary': False, 'min': 0, 'fiducial
↪': 1.0, 'prior_name': 'normal', 'mu': 1.0, 'sigma': 0.75, 'analytic': False}
theory.f1h_sBsB = {'name': 'f1h_sBsB', 'value': 4.0, 'vary': True, 'min': 0.0,
↪'fiducial': 4.0, 'prior_name': 'normal', 'mu': 4.0, 'sigma': 1.0, 'analytic': False}
theory.f_so = {'name': 'f_so', 'value': 0.03, 'vary': False, 'fiducial': 0.03, 'prior_
↪name': 'normal', 'mu': 0.04, 'sigma': 0.02, 'analytic': False}
theory.fcB = {'name': 'fcB', 'value': 0.08898809523809524, 'vary': False, 'min': 0,
↪'max': 1, 'expr': 'fs / (1 - fs) * (1 + fsB*(1./Nsat_mult - 1))', 'fiducial': 0.089,
↪ 'analytic': False}
theory.fs = {'name': 'fs', 'value': 0.104, 'vary': True, 'min': 0.0, 'max': 1.0,
↪'fiducial': 0.104, 'prior_name': 'uniform', 'lower': 0.0, 'upper': 0.25, 'analytic
↪': False}
```

(continues on next page)

(continued from previous page)

```
theory.fsB = {'name': 'fsB', 'value': 0.4, 'vary': True, 'min': 0.0, 'max': 1,
↪'fiducial': 0.4, 'prior_name': 'uniform', 'lower': 0.0, 'upper': 1.0, 'analytic':␣
↪False}
theory.fsigma8 = {'name': 'fsigma8', 'value': 0.4758, 'vary': False, 'expr':
↪'f*sigma8_z', 'analytic': False}
theory.gamma_b1cB = {'name': 'gamma_b1cB', 'value': 0.4, 'vary': False, 'min': 0.0,
↪'max': 1.0, 'fiducial': 0.4, 'prior_name': 'normal', 'mu': 0.4, 'sigma': 0.2,
↪'analytic': False}
theory.gamma_b1sA = {'name': 'gamma_b1sA', 'value': 1.45, 'vary': True, 'min': 1.0,
↪'fiducial': 1.45, 'prior_name': 'normal', 'mu': 1.45, 'sigma': 0.3, 'analytic':␣
↪False}
theory.gamma_b1sB = {'name': 'gamma_b1sB', 'value': 2.05, 'vary': True, 'min': 1.0,
↪'fiducial': 2.05, 'prior_name': 'normal', 'mu': 2.05, 'sigma': 0.3, 'analytic':␣
↪False}
theory.nbar = {'name': 'nbar', 'value': 0.0003117, 'vary': False, 'fiducial': 0.
↪0003117, 'analytic': False}
theory.sigma8_z = {'name': 'sigma8_z', 'value': 0.61, 'vary': True, 'fiducial': 0.61,
↪'prior_name': 'uniform', 'lower': 0.3, 'upper': 0.9, 'analytic': False}
theory.sigma_c = {'name': 'sigma_c', 'value': 1.0, 'vary': True, 'fiducial': 1.0,
↪'prior_name': 'uniform', 'lower': 0.0, 'upper': 3.0, 'analytic': False}
theory.sigma_sA = {'name': 'sigma_sA', 'value': 3.5, 'vary': True, 'fiducial': 3.5,
↪'prior_name': 'uniform', 'lower': 2.0, 'upper': 8.0, 'analytic': False}
theory.sigma_sB = {'name': 'sigma_sB', 'value': 5.0, 'vary': False, 'fiducial': 5.0,
↪'prior_name': 'uniform', 'lower': 3.0, 'upper': 10.0, 'analytic': False}
theory.sigma_so = {'name': 'sigma_so', 'value': 4.0, 'vary': False, 'fiducial': 4.0,
↪'prior_name': 'uniform', 'lower': 1.0, 'upper': 7, 'analytic': False}
#------------------------------------------------------------------------------

#------------------------------------------------------------------------------
# theory params
#------------------------------------------------------------------------------
theory.F_AP = {'name': 'F_AP', 'value': 1.0, 'vary': False, 'expr': 'alpha_par/alpha_
↪perp', 'analytic': False}
theory.N = {'name': 'N', 'value': 0.0, 'vary': False, 'fiducial': 0.0, 'prior_name':
↪'uniform', 'lower': -500.0, 'upper': 500.0, 'analytic': False}
theory.NcBs = {'name': 'NcBs', 'value': 40236.785434927464, 'vary': False, 'expr':
↪'f1h_cBs / (fcB*(1 - fs)*nbar)', 'fiducial': 45000.0, 'analytic': False}
theory.NsBsB = {'name': 'NsBsB', 'value': 128534.1756949072, 'vary': False, 'expr':
↪'f1h_sBsB / (fsB**2 * fs**2 * nbar) * (fcB*(1 - fs) - fs*(1-fsB))', 'fiducial':␣
↪94500.0, 'analytic': False}
theory.Nsat_mult = {'name': 'Nsat_mult', 'value': 2.4, 'vary': True, 'min': 2.0,
↪'fiducial': 2.4, 'prior_name': 'normal', 'mu': 2.4, 'sigma': 0.2, 'analytic': False}
theory.alpha = {'name': 'alpha', 'value': 1.0, 'vary': False, 'expr': '(alpha_perp**2␣
↪* alpha_par)**(1./3)', 'analytic': False}
theory.alpha_drag = {'name': 'alpha_drag', 'value': 1.0, 'vary': False, 'fiducial': 1.
↪0, 'analytic': False}
theory.alpha_par = {'name': 'alpha_par', 'value': 1.0, 'vary': True, 'fiducial': 1.0,
↪'prior_name': 'uniform', 'lower': 0.8, 'upper': 1.2, 'analytic': False}
theory.alpha_perp = {'name': 'alpha_perp', 'value': 1.0, 'vary': True, 'fiducial': 1.
↪0, 'prior_name': 'uniform', 'lower': 0.8, 'upper': 1.2, 'analytic': False}
theory.b1 = {'name': 'b1', 'value': 2.124276, 'vary': False, 'expr': '(1 - fs)*b1_c +␣
↪fs*b1_s', 'analytic': False}
theory.b1_c = {'name': 'b1_c', 'value': 1.9981383928571428, 'vary': False, 'expr':
↪'(1 - fcB)*b1_cA + fcB*b1_cB', 'analytic': False}
theory.b1_cA = {'name': 'b1_cA', 'value': 1.9, 'vary': True, 'fiducial': 1.9, 'prior_
↪name': 'uniform', 'lower': 1.2, 'upper': 2.5, 'analytic': False}
theory.b1_cB = {'name': 'b1_cB', 'value': 3.0028260869565218, 'vary': False, 'expr':
↪'(1-fsB)/(1+fsB*(1./Nsat_mult - 1)) * b1_sA +  (1 - (1-fsB)/(1+fsB*(1./Nsat_mult -␣
↪1))) * b1_sB', 'fiducial': 2.84, 'analytic': False}
```

(continues on next page)

(continued from previous page)

```
theory.b1_s = {'name': 'b1_s', 'value': 3.2109999999999994, 'vary': False, 'expr':
↪'(1 - fsB)*b1_sA + fsB*b1_sB', 'analytic': False}
theory.b1_sA = {'name': 'b1_sA', 'value': 2.755, 'vary': False, 'expr': 'gamma_
↪b1sA*b1_cA', 'fiducial': 2.63, 'analytic': False}
theory.b1_sB = {'name': 'b1_sB', 'value': 3.8949999999999996, 'vary': False, 'expr':
↪'gamma_b1sB*b1_cA', 'fiducial': 3.62, 'analytic': False}
theory.b1sigma8 = {'name': 'b1sigma8', 'value': 1.29580836, 'vary': False, 'expr':
↪'b1*sigma8_z', 'analytic': False}
theory.epsilon = {'name': 'epsilon', 'value': 0.0, 'vary': False, 'expr': '(alpha_
↪perp/alpha_par)**(-1./3) - 1.0', 'analytic': False}
theory.f = {'name': 'f', 'value': 0.78, 'vary': True, 'fiducial': 0.78, 'prior_name':
↪'uniform', 'lower': 0.6, 'upper': 1.0, 'analytic': False}
theory.f1h_cBs = {'name': 'f1h_cBs', 'value': 1.0, 'vary': False, 'min': 0, 'fiducial
↪': 1.0, 'prior_name': 'normal', 'mu': 1.0, 'sigma': 0.75, 'analytic': False}
theory.f1h_sBsB = {'name': 'f1h_sBsB', 'value': 4.0, 'vary': True, 'min': 0.0,
↪'fiducial': 4.0, 'prior_name': 'normal', 'mu': 4.0, 'sigma': 1.0, 'analytic': False}
theory.f_so = {'name': 'f_so', 'value': 0.03, 'vary': False, 'fiducial': 0.03, 'prior_
↪name': 'normal', 'mu': 0.04, 'sigma': 0.02, 'analytic': False}
theory.fcB = {'name': 'fcB', 'value': 0.08898809523809524, 'vary': False, 'min': 0,
↪'max': 1, 'expr': 'fs / (1 - fs) * (1 + fsB*(1./Nsat_mult - 1))', 'fiducial': 0.089,
↪ 'analytic': False}
theory.fs = {'name': 'fs', 'value': 0.104, 'vary': True, 'min': 0.0, 'max': 1.0,
↪'fiducial': 0.104, 'prior_name': 'uniform', 'lower': 0.0, 'upper': 0.25, 'analytic
↪': False}
theory.fsB = {'name': 'fsB', 'value': 0.4, 'vary': True, 'min': 0.0, 'max': 1,
↪'fiducial': 0.4, 'prior_name': 'uniform', 'lower': 0.0, 'upper': 1.0, 'analytic':␣
↪False}
theory.fsigma8 = {'name': 'fsigma8', 'value': 0.4758, 'vary': False, 'expr':
↪'f*sigma8_z', 'analytic': False}
theory.gamma_b1cB = {'name': 'gamma_b1cB', 'value': 0.4, 'vary': False, 'min': 0.0,
↪'max': 1.0, 'fiducial': 0.4, 'prior_name': 'normal', 'mu': 0.4, 'sigma': 0.2,
↪'analytic': False}
theory.gamma_b1sA = {'name': 'gamma_b1sA', 'value': 1.45, 'vary': True, 'min': 1.0,
↪'fiducial': 1.45, 'prior_name': 'normal', 'mu': 1.45, 'sigma': 0.3, 'analytic':␣
↪False}
theory.gamma_b1sB = {'name': 'gamma_b1sB', 'value': 2.05, 'vary': True, 'min': 1.0,
↪'fiducial': 2.05, 'prior_name': 'normal', 'mu': 2.05, 'sigma': 0.3, 'analytic':␣
↪False}
theory.nbar = {'name': 'nbar', 'value': 0.0003117, 'vary': False, 'fiducial': 0.
↪0003117, 'analytic': False}
theory.sigma8_z = {'name': 'sigma8_z', 'value': 0.61, 'vary': True, 'fiducial': 0.61,
↪'prior_name': 'uniform', 'lower': 0.3, 'upper': 0.9, 'analytic': False}
theory.sigma_c = {'name': 'sigma_c', 'value': 1.0, 'vary': True, 'fiducial': 1.0,
↪'prior_name': 'uniform', 'lower': 0.0, 'upper': 3.0, 'analytic': False}
theory.sigma_sA = {'name': 'sigma_sA', 'value': 3.5, 'vary': True, 'fiducial': 3.5,
↪'prior_name': 'uniform', 'lower': 2.0, 'upper': 8.0, 'analytic': False}
theory.sigma_sB = {'name': 'sigma_sB', 'value': 5.0, 'vary': False, 'fiducial': 5.0,
↪'prior_name': 'uniform', 'lower': 3.0, 'upper': 10.0, 'analytic': False}
theory.sigma_so = {'name': 'sigma_so', 'value': 4.0, 'vary': False, 'fiducial': 4.0,
↪'prior_name': 'uniform', 'lower': 1.0, 'upper': 7, 'analytic': False}
#------------------------------------------------------------------------------

#------------------------------------------------------------------------------
# theory params
#------------------------------------------------------------------------------
theory.F_AP = {'name': 'F_AP', 'value': 1.0, 'vary': False, 'expr': 'alpha_par/alpha_
↪perp', 'analytic': False}
```

(continues on next page)

(continued from previous page)

```
theory.N = {'name': 'N', 'value': 0.0, 'vary': False, 'fiducial': 0.0, 'prior_name':
↪'uniform', 'lower': -500.0, 'upper': 500.0, 'analytic': False}
theory.NcBs = {'name': 'NcBs', 'value': 40236.785434927464, 'vary': False, 'expr':
↪'f1h_cBs / (fcB*(1 - fs)*nbar)', 'fiducial': 45000.0, 'analytic': False}
theory.NsBsB = {'name': 'NsBsB', 'value': 128534.1756949072, 'vary': False, 'expr':
↪'f1h_sBsB / (fsB**2 * fs**2 * nbar) * (fcB*(1 - fs) - fs*(1-fsB))', 'fiducial':␣
↪94500.0, 'analytic': False}
theory.Nsat_mult = {'name': 'Nsat_mult', 'value': 2.4, 'vary': True, 'min': 2.0,
↪'fiducial': 2.4, 'prior_name': 'normal', 'mu': 2.4, 'sigma': 0.2, 'analytic': False}
theory.alpha = {'name': 'alpha', 'value': 1.0, 'vary': False, 'expr': '(alpha_perp**2␣
↪* alpha_par)**(1./3)', 'analytic': False}
theory.alpha_drag = {'name': 'alpha_drag', 'value': 1.0, 'vary': False, 'fiducial': 1.
↪0, 'analytic': False}
theory.alpha_par = {'name': 'alpha_par', 'value': 1.0, 'vary': True, 'fiducial': 1.0,
↪'prior_name': 'uniform', 'lower': 0.8, 'upper': 1.2, 'analytic': False}
theory.alpha_perp = {'name': 'alpha_perp', 'value': 1.0, 'vary': True, 'fiducial': 1.
↪0, 'prior_name': 'uniform', 'lower': 0.8, 'upper': 1.2, 'analytic': False}
theory.b1 = {'name': 'b1', 'value': 2.124276, 'vary': False, 'expr': '(1 - fs)*b1_c +␣
↪fs*b1_s', 'analytic': False}
theory.b1_c = {'name': 'b1_c', 'value': 1.9981383928571428, 'vary': False, 'expr':
↪'(1 - fcB)*b1_cA + fcB*b1_cB', 'analytic': False}
theory.b1_cA = {'name': 'b1_cA', 'value': 1.9, 'vary': True, 'fiducial': 1.9, 'prior_
↪name': 'uniform', 'lower': 1.2, 'upper': 2.5, 'analytic': False}
theory.b1_cB = {'name': 'b1_cB', 'value': 3.0028260869565218, 'vary': False, 'expr':
↪'(1-fsB)/(1+fsB*(1./Nsat_mult - 1)) * b1_sA +  (1 - (1-fsB)/(1+fsB*(1./Nsat_mult -␣
↪1))) * b1_sB', 'fiducial': 2.84, 'analytic': False}
theory.b1_s = {'name': 'b1_s', 'value': 3.2109999999999994, 'vary': False, 'expr':
↪'(1 - fsB)*b1_sA + fsB*b1_sB', 'analytic': False}
theory.b1_sA = {'name': 'b1_sA', 'value': 2.755, 'vary': False, 'expr': 'gamma_
↪b1sA*b1_cA', 'fiducial': 2.63, 'analytic': False}
theory.b1_sB = {'name': 'b1_sB', 'value': 3.8949999999999996, 'vary': False, 'expr':
↪'gamma_b1sB*b1_cA', 'fiducial': 3.62, 'analytic': False}
theory.b1sigma8 = {'name': 'b1sigma8', 'value': 1.29580836, 'vary': False, 'expr':
↪'b1*sigma8_z', 'analytic': False}
theory.epsilon = {'name': 'epsilon', 'value': 0.0, 'vary': False, 'expr': '(alpha_
↪perp/alpha_par)**(-1./3) - 1.0', 'analytic': False}
theory.f = {'name': 'f', 'value': 0.78, 'vary': True, 'fiducial': 0.78, 'prior_name':
↪'uniform', 'lower': 0.6, 'upper': 1.0, 'analytic': False}
theory.f1h_cBs = {'name': 'f1h_cBs', 'value': 1.0, 'vary': False, 'min': 0, 'fiducial
↪': 1.0, 'prior_name': 'normal', 'mu': 1.0, 'sigma': 0.75, 'analytic': False}
theory.f1h_sBsB = {'name': 'f1h_sBsB', 'value': 4.0, 'vary': True, 'min': 0.0,
↪'fiducial': 4.0, 'prior_name': 'normal', 'mu': 4.0, 'sigma': 1.0, 'analytic': False}
theory.f_so = {'name': 'f_so', 'value': 0.03, 'vary': False, 'fiducial': 0.03, 'prior_
↪name': 'normal', 'mu': 0.04, 'sigma': 0.02, 'analytic': False}
theory.fcB = {'name': 'fcB', 'value': 0.08898809523809524, 'vary': False, 'min': 0,
↪'max': 1, 'expr': 'fs / (1 - fs) * (1 + fsB*(1./Nsat_mult - 1))', 'fiducial': 0.089,
↪ 'analytic': False}
theory.fs = {'name': 'fs', 'value': 0.104, 'vary': True, 'min': 0.0, 'max': 1.0,
↪'fiducial': 0.104, 'prior_name': 'uniform', 'lower': 0.0, 'upper': 0.25, 'analytic
↪': False}
theory.fsB = {'name': 'fsB', 'value': 0.4, 'vary': True, 'min': 0.0, 'max': 1,
↪'fiducial': 0.4, 'prior_name': 'uniform', 'lower': 0.0, 'upper': 1.0, 'analytic':␣
↪False}
theory.fsigma8 = {'name': 'fsigma8', 'value': 0.4758, 'vary': False, 'expr':
↪'f*sigma8_z', 'analytic': False}
theory.gamma_b1cB = {'name': 'gamma_b1cB', 'value': 0.4, 'vary': False, 'min': 0.0,
↪'max': 1.0, 'fiducial': 0.4, 'prior_name': 'normal', 'mu': 0.4, 'sigma': 0.2,
↪'analytic': False}
```

(continues on next page)

(continued from previous page)

```
theory.gamma_b1sA = {'name': 'gamma_b1sA', 'value': 1.45, 'vary': True, 'min': 1.0,
↪'fiducial': 1.45, 'prior_name': 'normal', 'mu': 1.45, 'sigma': 0.3, 'analytic':␣
↪False}
theory.gamma_b1sB = {'name': 'gamma_b1sB', 'value': 2.05, 'vary': True, 'min': 1.0,
↪'fiducial': 2.05, 'prior_name': 'normal', 'mu': 2.05, 'sigma': 0.3, 'analytic':␣
↪False}
theory.nbar = {'name': 'nbar', 'value': 0.0003117, 'vary': False, 'fiducial': 0.
↪0003117, 'analytic': False}
theory.sigma8_z = {'name': 'sigma8_z', 'value': 0.61, 'vary': True, 'fiducial': 0.61,
↪'prior_name': 'uniform', 'lower': 0.3, 'upper': 0.9, 'analytic': False}
theory.sigma_c = {'name': 'sigma_c', 'value': 1.0, 'vary': True, 'fiducial': 1.0,
↪'prior_name': 'uniform', 'lower': 0.0, 'upper': 3.0, 'analytic': False}
theory.sigma_sA = {'name': 'sigma_sA', 'value': 3.5, 'vary': True, 'fiducial': 3.5,
↪'prior_name': 'uniform', 'lower': 2.0, 'upper': 8.0, 'analytic': False}
theory.sigma_sB = {'name': 'sigma_sB', 'value': 5.0, 'vary': False, 'fiducial': 5.0,
↪'prior_name': 'uniform', 'lower': 3.0, 'upper': 10.0, 'analytic': False}
theory.sigma_so = {'name': 'sigma_so', 'value': 4.0, 'vary': False, 'fiducial': 4.0,
↪'prior_name': 'uniform', 'lower': 1.0, 'upper': 7, 'analytic': False}
#-------------------------------------------------------------------------------

#-------------------------------------------------------------------------------
# theory params
#-------------------------------------------------------------------------------
theory.F_AP = {'name': 'F_AP', 'value': 1.0, 'vary': False, 'expr': 'alpha_par/alpha_
↪perp', 'analytic': False}
theory.N = {'name': 'N', 'value': 0.0, 'vary': False, 'fiducial': 0.0, 'prior_name':
↪'uniform', 'lower': -500.0, 'upper': 500.0, 'analytic': False}
theory.NcBs = {'name': 'NcBs', 'value': 40236.785434927464, 'vary': False, 'expr':
↪'f1h_cBs / (fcB*(1 - fs)*nbar)', 'fiducial': 45000.0, 'analytic': False}
theory.NsBsB = {'name': 'NsBsB', 'value': 128534.1756949072, 'vary': False, 'expr':
↪'f1h_sBsB / (fsB**2 * fs**2 * nbar) * (fcB*(1 - fs) - fs*(1-fsB))', 'fiducial':␣
↪94500.0, 'analytic': False}
theory.Nsat_mult = {'name': 'Nsat_mult', 'value': 2.4, 'vary': True, 'min': 2.0,
↪'fiducial': 2.4, 'prior_name': 'normal', 'mu': 2.4, 'sigma': 0.2, 'analytic': False}
theory.alpha = {'name': 'alpha', 'value': 1.0, 'vary': False, 'expr': '(alpha_perp**2␣
↪* alpha_par)**(1./3)', 'analytic': False}
theory.alpha_drag = {'name': 'alpha_drag', 'value': 1.0, 'vary': False, 'fiducial': 1.
↪0, 'analytic': False}
theory.alpha_par = {'name': 'alpha_par', 'value': 1.0, 'vary': True, 'fiducial': 1.0,
↪'prior_name': 'uniform', 'lower': 0.8, 'upper': 1.2, 'analytic': False}
theory.alpha_perp = {'name': 'alpha_perp', 'value': 1.0, 'vary': True, 'fiducial': 1.
↪0, 'prior_name': 'uniform', 'lower': 0.8, 'upper': 1.2, 'analytic': False}
theory.b1 = {'name': 'b1', 'value': 2.124276, 'vary': False, 'expr': '(1 - fs)*b1_c +␣
↪fs*b1_s', 'analytic': False}
theory.b1_c = {'name': 'b1_c', 'value': 1.9981383928571428, 'vary': False, 'expr':
↪'(1 - fcB)*b1_cA + fcB*b1_cB', 'analytic': False}
theory.b1_cA = {'name': 'b1_cA', 'value': 1.9, 'vary': True, 'fiducial': 1.9, 'prior_
↪name': 'uniform', 'lower': 1.2, 'upper': 2.5, 'analytic': False}
theory.b1_cB = {'name': 'b1_cB', 'value': 3.0028260869565218, 'vary': False, 'expr':
↪'(1-fsB)/(1+fsB*(1./Nsat_mult - 1)) * b1_sA +  (1 - (1-fsB)/(1+fsB*(1./Nsat_mult -␣
↪1))) * b1_sB', 'fiducial': 2.84, 'analytic': False}
theory.b1_s = {'name': 'b1_s', 'value': 3.2109999999999994, 'vary': False, 'expr':
↪'(1 - fsB)*b1_sA + fsB*b1_sB', 'analytic': False}
theory.b1_sA = {'name': 'b1_sA', 'value': 2.755, 'vary': False, 'expr': 'gamma_
↪b1sA*b1_cA', 'fiducial': 2.63, 'analytic': False}
theory.b1_sB = {'name': 'b1_sB', 'value': 3.8949999999999996, 'vary': False, 'expr':
↪'gamma_b1sB*b1_cA', 'fiducial': 3.62, 'analytic': False}
```

(continues on next page)

(continued from previous page)

```
theory.b1sigma8 = {'name': 'b1sigma8', 'value': 1.29580836, 'vary': False, 'expr':
↪'b1*sigma8_z', 'analytic': False}
theory.epsilon = {'name': 'epsilon', 'value': 0.0, 'vary': False, 'expr': '(alpha_
↪perp/alpha_par)**(-1./3) - 1.0', 'analytic': False}
theory.f = {'name': 'f', 'value': 0.78, 'vary': True, 'fiducial': 0.78, 'prior_name':
↪'uniform', 'lower': 0.6, 'upper': 1.0, 'analytic': False}
theory.f1h_cBs = {'name': 'f1h_cBs', 'value': 1.0, 'vary': False, 'min': 0, 'fiducial
↪': 1.0, 'prior_name': 'normal', 'mu': 1.0, 'sigma': 0.75, 'analytic': False}
theory.f1h_sBsB = {'name': 'f1h_sBsB', 'value': 4.0, 'vary': True, 'min': 0.0,
↪'fiducial': 4.0, 'prior_name': 'normal', 'mu': 4.0, 'sigma': 1.0, 'analytic': False}
theory.f_so = {'name': 'f_so', 'value': 0.03, 'vary': False, 'fiducial': 0.03, 'prior_
↪name': 'normal', 'mu': 0.04, 'sigma': 0.02, 'analytic': False}
theory.fcB = {'name': 'fcB', 'value': 0.08898809523809524, 'vary': False, 'min': 0,
↪'max': 1, 'expr': 'fs / (1 - fs) * (1 + fsB*(1./Nsat_mult - 1))', 'fiducial': 0.089,
↪ 'analytic': False}
theory.fs = {'name': 'fs', 'value': 0.104, 'vary': True, 'min': 0.0, 'max': 1.0,
↪'fiducial': 0.104, 'prior_name': 'uniform', 'lower': 0.0, 'upper': 0.25, 'analytic
↪': False}
theory.fsB = {'name': 'fsB', 'value': 0.4, 'vary': True, 'min': 0.0, 'max': 1,
↪'fiducial': 0.4, 'prior_name': 'uniform', 'lower': 0.0, 'upper': 1.0, 'analytic':␣
↪False}
theory.fsigma8 = {'name': 'fsigma8', 'value': 0.4758, 'vary': False, 'expr':
↪'f*sigma8_z', 'analytic': False}
theory.gamma_b1cB = {'name': 'gamma_b1cB', 'value': 0.4, 'vary': False, 'min': 0.0,
↪'max': 1.0, 'fiducial': 0.4, 'prior_name': 'normal', 'mu': 0.4, 'sigma': 0.2,
↪'analytic': False}
theory.gamma_b1sA = {'name': 'gamma_b1sA', 'value': 1.45, 'vary': True, 'min': 1.0,
↪'fiducial': 1.45, 'prior_name': 'normal', 'mu': 1.45, 'sigma': 0.3, 'analytic':␣
↪False}
theory.gamma_b1sB = {'name': 'gamma_b1sB', 'value': 2.05, 'vary': True, 'min': 1.0,
↪'fiducial': 2.05, 'prior_name': 'normal', 'mu': 2.05, 'sigma': 0.3, 'analytic':␣
↪False}
theory.nbar = {'name': 'nbar', 'value': 0.0003117, 'vary': False, 'fiducial': 0.
↪0003117, 'analytic': False}
theory.sigma8_z = {'name': 'sigma8_z', 'value': 0.61, 'vary': True, 'fiducial': 0.61,
↪'prior_name': 'uniform', 'lower': 0.3, 'upper': 0.9, 'analytic': False}
theory.sigma_c = {'name': 'sigma_c', 'value': 1.0, 'vary': True, 'fiducial': 1.0,
↪'prior_name': 'uniform', 'lower': 0.0, 'upper': 3.0, 'analytic': False}
theory.sigma_sA = {'name': 'sigma_sA', 'value': 3.5, 'vary': True, 'fiducial': 3.5,
↪'prior_name': 'uniform', 'lower': 2.0, 'upper': 8.0, 'analytic': False}
theory.sigma_sB = {'name': 'sigma_sB', 'value': 5.0, 'vary': False, 'fiducial': 5.0,
↪'prior_name': 'uniform', 'lower': 3.0, 'upper': 10.0, 'analytic': False}
theory.sigma_so = {'name': 'sigma_so', 'value': 4.0, 'vary': False, 'fiducial': 4.0,
↪'prior_name': 'uniform', 'lower': 1.0, 'upper': 7, 'analytic': False}
#------------------------------------------------------------------------------

#------------------------------------------------------------------------------
# theory params
#------------------------------------------------------------------------------
theory.F_AP = {'name': 'F_AP', 'value': 1.0, 'vary': False, 'expr': 'alpha_par/alpha_
↪perp', 'analytic': False}
theory.N = {'name': 'N', 'value': 0.0, 'vary': False, 'fiducial': 0.0, 'prior_name':
↪'uniform', 'lower': -500.0, 'upper': 500.0, 'analytic': False}
theory.NcBs = {'name': 'NcBs', 'value': 40236.785434927464, 'vary': False, 'expr':
↪'f1h_cBs / (fcB*(1 - fs)*nbar)', 'fiducial': 45000.0, 'analytic': False}
theory.NsBsB = {'name': 'NsBsB', 'value': 128534.1756949072, 'vary': False, 'expr':
↪'f1h_sBsB / (fsB**2 * fs**2 * nbar) * (fcB*(1 - fs) - fs*(1-fsB))', 'fiducial':␣
↪94500.0, 'analytic': False}
```

(continues on next page)

(continued from previous page)

```
theory.Nsat_mult = {'name': 'Nsat_mult', 'value': 2.4, 'vary': True, 'min': 2.0,
↪'fiducial': 2.4, 'prior_name': 'normal', 'mu': 2.4, 'sigma': 0.2, 'analytic': False}
theory.alpha = {'name': 'alpha', 'value': 1.0, 'vary': False, 'expr': '(alpha_perp**2␣
↪* alpha_par)**(1./3)', 'analytic': False}
theory.alpha_drag = {'name': 'alpha_drag', 'value': 1.0, 'vary': False, 'fiducial': 1.
↪0, 'analytic': False}
theory.alpha_par = {'name': 'alpha_par', 'value': 1.0, 'vary': True, 'fiducial': 1.0,
↪'prior_name': 'uniform', 'lower': 0.8, 'upper': 1.2, 'analytic': False}
theory.alpha_perp = {'name': 'alpha_perp', 'value': 1.0, 'vary': True, 'fiducial': 1.
↪0, 'prior_name': 'uniform', 'lower': 0.8, 'upper': 1.2, 'analytic': False}
theory.b1 = {'name': 'b1', 'value': 2.124276, 'vary': False, 'expr': '(1 - fs)*b1_c +␣
↪fs*b1_s', 'analytic': False}
theory.b1_c = {'name': 'b1_c', 'value': 1.9981383928571428, 'vary': False, 'expr':
↪'(1 - fcB)*b1_cA + fcB*b1_cB', 'analytic': False}
theory.b1_cA = {'name': 'b1_cA', 'value': 1.9, 'vary': True, 'fiducial': 1.9, 'prior_
↪name': 'uniform', 'lower': 1.2, 'upper': 2.5, 'analytic': False}
theory.b1_cB = {'name': 'b1_cB', 'value': 3.0028260869565218, 'vary': False, 'expr':
↪'(1-fsB)/(1+fsB*(1./Nsat_mult - 1)) * b1_sA +  (1 - (1-fsB)/(1+fsB*(1./Nsat_mult -␣
↪1))) * b1_sB', 'fiducial': 2.84, 'analytic': False}
theory.b1_s = {'name': 'b1_s', 'value': 3.2109999999999994, 'vary': False, 'expr':
↪'(1 - fsB)*b1_sA + fsB*b1_sB', 'analytic': False}
theory.b1_sA = {'name': 'b1_sA', 'value': 2.755, 'vary': False, 'expr': 'gamma_
↪b1sA*b1_cA', 'fiducial': 2.63, 'analytic': False}
theory.b1_sB = {'name': 'b1_sB', 'value': 3.8949999999999996, 'vary': False, 'expr':
↪'gamma_b1sB*b1_cA', 'fiducial': 3.62, 'analytic': False}
theory.b1sigma8 = {'name': 'b1sigma8', 'value': 1.29580836, 'vary': False, 'expr':
↪'b1*sigma8_z', 'analytic': False}
theory.epsilon = {'name': 'epsilon', 'value': 0.0, 'vary': False, 'expr': '(alpha_
↪perp/alpha_par)**(-1./3) - 1.0', 'analytic': False}
theory.f = {'name': 'f', 'value': 0.78, 'vary': True, 'fiducial': 0.78, 'prior_name':
↪'uniform', 'lower': 0.6, 'upper': 1.0, 'analytic': False}
theory.f1h_cBs = {'name': 'f1h_cBs', 'value': 1.0, 'vary': False, 'min': 0, 'fiducial
↪': 1.0, 'prior_name': 'normal', 'mu': 1.0, 'sigma': 0.75, 'analytic': False}
theory.f1h_sBsB = {'name': 'f1h_sBsB', 'value': 4.0, 'vary': True, 'min': 0.0,
↪'fiducial': 4.0, 'prior_name': 'normal', 'mu': 4.0, 'sigma': 1.0, 'analytic': False}
theory.f_so = {'name': 'f_so', 'value': 0.03, 'vary': False, 'fiducial': 0.03, 'prior_
↪name': 'normal', 'mu': 0.04, 'sigma': 0.02, 'analytic': False}
theory.fcB = {'name': 'fcB', 'value': 0.08898809523809524, 'vary': False, 'min': 0,
↪'max': 1, 'expr': 'fs / (1 - fs) * (1 + fsB*(1./Nsat_mult - 1))', 'fiducial': 0.089,
↪ 'analytic': False}
theory.fs = {'name': 'fs', 'value': 0.104, 'vary': True, 'min': 0.0, 'max': 1.0,
↪'fiducial': 0.104, 'prior_name': 'uniform', 'lower': 0.0, 'upper': 0.25, 'analytic
↪': False}
theory.fsB = {'name': 'fsB', 'value': 0.4, 'vary': True, 'min': 0.0, 'max': 1,
↪'fiducial': 0.4, 'prior_name': 'uniform', 'lower': 0.0, 'upper': 1.0, 'analytic':␣
↪False}
theory.fsigma8 = {'name': 'fsigma8', 'value': 0.4758, 'vary': False, 'expr':
↪'f*sigma8_z', 'analytic': False}
theory.gamma_b1cB = {'name': 'gamma_b1cB', 'value': 0.4, 'vary': False, 'min': 0.0,
↪'max': 1.0, 'fiducial': 0.4, 'prior_name': 'normal', 'mu': 0.4, 'sigma': 0.2,
↪'analytic': False}
theory.gamma_b1sA = {'name': 'gamma_b1sA', 'value': 1.45, 'vary': True, 'min': 1.0,
↪'fiducial': 1.45, 'prior_name': 'normal', 'mu': 1.45, 'sigma': 0.3, 'analytic':␣
↪False}
theory.gamma_b1sB = {'name': 'gamma_b1sB', 'value': 2.05, 'vary': True, 'min': 1.0,
↪'fiducial': 2.05, 'prior_name': 'normal', 'mu': 2.05, 'sigma': 0.3, 'analytic':␣
↪False}
```

(continues on next page)

(continued from previous page)

```
theory.nbar = {'name': 'nbar', 'value': 0.0003117, 'vary': False, 'fiducial': 0.
↪0003117, 'analytic': False}
theory.sigma8_z = {'name': 'sigma8_z', 'value': 0.61, 'vary': True, 'fiducial': 0.61,
↪'prior_name': 'uniform', 'lower': 0.3, 'upper': 0.9, 'analytic': False}
theory.sigma_c = {'name': 'sigma_c', 'value': 1.0, 'vary': True, 'fiducial': 1.0,
↪'prior_name': 'uniform', 'lower': 0.0, 'upper': 3.0, 'analytic': False}
theory.sigma_sA = {'name': 'sigma_sA', 'value': 3.5, 'vary': True, 'fiducial': 3.5,
↪'prior_name': 'uniform', 'lower': 2.0, 'upper': 8.0, 'analytic': False}
theory.sigma_sB = {'name': 'sigma_sB', 'value': 5.0, 'vary': False, 'fiducial': 5.0,
↪'prior_name': 'uniform', 'lower': 3.0, 'upper': 10.0, 'analytic': False}
theory.sigma_so = {'name': 'sigma_so', 'value': 4.0, 'vary': False, 'fiducial': 4.0,
↪'prior_name': 'uniform', 'lower': 1.0, 'upper': 7, 'analytic': False}
#------------------------------------------------------------------------------

#------------------------------------------------------------------------------
# theory params
#------------------------------------------------------------------------------
theory.F_AP = {'name': 'F_AP', 'value': 1.0, 'vary': False, 'expr': 'alpha_par/alpha_
↪perp', 'analytic': False}
theory.N = {'name': 'N', 'value': 0.0, 'vary': False, 'fiducial': 0.0, 'prior_name':
↪'uniform', 'lower': -500.0, 'upper': 500.0, 'analytic': False}
theory.NcBs = {'name': 'NcBs', 'value': 40236.785434927464, 'vary': False, 'expr':
↪'f1h_cBs / (fcB*(1 - fs)*nbar)', 'fiducial': 45000.0, 'analytic': False}
theory.NsBsB = {'name': 'NsBsB', 'value': 128534.1756949072, 'vary': False, 'expr':
↪'f1h_sBsB / (fsB**2 * fs**2 * nbar) * (fcB*(1 - fs) - fs*(1-fsB))', 'fiducial':␣
↪94500.0, 'analytic': False}
theory.Nsat_mult = {'name': 'Nsat_mult', 'value': 2.4, 'vary': True, 'min': 2.0,
↪'fiducial': 2.4, 'prior_name': 'normal', 'mu': 2.4, 'sigma': 0.2, 'analytic': False}
theory.alpha = {'name': 'alpha', 'value': 1.0, 'vary': False, 'expr': '(alpha_perp**2␣
↪* alpha_par)**(1./3)', 'analytic': False}
theory.alpha_drag = {'name': 'alpha_drag', 'value': 1.0, 'vary': False, 'fiducial': 1.
↪0, 'analytic': False}
theory.alpha_par = {'name': 'alpha_par', 'value': 1.0, 'vary': True, 'fiducial': 1.0,
↪'prior_name': 'uniform', 'lower': 0.8, 'upper': 1.2, 'analytic': False}
theory.alpha_perp = {'name': 'alpha_perp', 'value': 1.0, 'vary': True, 'fiducial': 1.
↪0, 'prior_name': 'uniform', 'lower': 0.8, 'upper': 1.2, 'analytic': False}
theory.b1 = {'name': 'b1', 'value': 2.124276, 'vary': False, 'expr': '(1 - fs)*b1_c +␣
↪fs*b1_s', 'analytic': False}
theory.b1_c = {'name': 'b1_c', 'value': 1.9981383928571428, 'vary': False, 'expr':
↪'(1 - fcB)*b1_cA + fcB*b1_cB', 'analytic': False}
theory.b1_cA = {'name': 'b1_cA', 'value': 1.9, 'vary': True, 'fiducial': 1.9, 'prior_
↪name': 'uniform', 'lower': 1.2, 'upper': 2.5, 'analytic': False}
theory.b1_cB = {'name': 'b1_cB', 'value': 3.0028260869565218, 'vary': False, 'expr':
↪'(1-fsB)/(1+fsB*(1./Nsat_mult - 1)) * b1_sA +  (1 - (1-fsB)/(1+fsB*(1./Nsat_mult -␣
↪1))) * b1_sB', 'fiducial': 2.84, 'analytic': False}
theory.b1_s = {'name': 'b1_s', 'value': 3.2109999999999994, 'vary': False, 'expr':
↪'(1 - fsB)*b1_sA + fsB*b1_sB', 'analytic': False}
theory.b1_sA = {'name': 'b1_sA', 'value': 2.755, 'vary': False, 'expr': 'gamma_
↪b1sA*b1_cA', 'fiducial': 2.63, 'analytic': False}
theory.b1_sB = {'name': 'b1_sB', 'value': 3.8949999999999996, 'vary': False, 'expr':
↪'gamma_b1sB*b1_cA', 'fiducial': 3.62, 'analytic': False}
theory.b1sigma8 = {'name': 'b1sigma8', 'value': 1.29580836, 'vary': False, 'expr':
↪'b1*sigma8_z', 'analytic': False}
theory.epsilon = {'name': 'epsilon', 'value': 0.0, 'vary': False, 'expr': '(alpha_
↪perp/alpha_par)**(-1./3) - 1.0', 'analytic': False}
theory.f = {'name': 'f', 'value': 0.78, 'vary': True, 'fiducial': 0.78, 'prior_name':
↪'uniform', 'lower': 0.6, 'upper': 1.0, 'analytic': False}
```

(continues on next page)

(continued from previous page)

```
theory.f1h_cBs = {'name': 'f1h_cBs', 'value': 1.0, 'vary': False, 'min': 0, 'fiducial
↪': 1.0, 'prior_name': 'normal', 'mu': 1.0, 'sigma': 0.75, 'analytic': False}
theory.f1h_sBsB = {'name': 'f1h_sBsB', 'value': 4.0, 'vary': True, 'min': 0.0,
↪'fiducial': 4.0, 'prior_name': 'normal', 'mu': 4.0, 'sigma': 1.0, 'analytic': False}
theory.f_so = {'name': 'f_so', 'value': 0.03, 'vary': False, 'fiducial': 0.03, 'prior_
↪name': 'normal', 'mu': 0.04, 'sigma': 0.02, 'analytic': False}
theory.fcB = {'name': 'fcB', 'value': 0.08898809523809524, 'vary': False, 'min': 0,
↪'max': 1, 'expr': 'fs / (1 - fs) * (1 + fsB*(1./Nsat_mult - 1))', 'fiducial': 0.089,
↪ 'analytic': False}
theory.fs = {'name': 'fs', 'value': 0.104, 'vary': True, 'min': 0.0, 'max': 1.0,
↪'fiducial': 0.104, 'prior_name': 'uniform', 'lower': 0.0, 'upper': 0.25, 'analytic
↪': False}
theory.fsB = {'name': 'fsB', 'value': 0.4, 'vary': True, 'min': 0.0, 'max': 1,
↪'fiducial': 0.4, 'prior_name': 'uniform', 'lower': 0.0, 'upper': 1.0, 'analytic':␣
↪False}
theory.fsigma8 = {'name': 'fsigma8', 'value': 0.4758, 'vary': False, 'expr':
↪'f*sigma8_z', 'analytic': False}
theory.gamma_b1cB = {'name': 'gamma_b1cB', 'value': 0.4, 'vary': False, 'min': 0.0,
↪'max': 1.0, 'fiducial': 0.4, 'prior_name': 'normal', 'mu': 0.4, 'sigma': 0.2,
↪'analytic': False}
theory.gamma_b1sA = {'name': 'gamma_b1sA', 'value': 1.45, 'vary': True, 'min': 1.0,
↪'fiducial': 1.45, 'prior_name': 'normal', 'mu': 1.45, 'sigma': 0.3, 'analytic':␣
↪False}
theory.gamma_b1sB = {'name': 'gamma_b1sB', 'value': 2.05, 'vary': True, 'min': 1.0,
↪'fiducial': 2.05, 'prior_name': 'normal', 'mu': 2.05, 'sigma': 0.3, 'analytic':␣
↪False}
theory.nbar = {'name': 'nbar', 'value': 0.0003117, 'vary': False, 'fiducial': 0.
↪0003117, 'analytic': False}
theory.sigma8_z = {'name': 'sigma8_z', 'value': 0.61, 'vary': True, 'fiducial': 0.61,
↪'prior_name': 'uniform', 'lower': 0.3, 'upper': 0.9, 'analytic': False}
theory.sigma_c = {'name': 'sigma_c', 'value': 1.0, 'vary': True, 'fiducial': 1.0,
↪'prior_name': 'uniform', 'lower': 0.0, 'upper': 3.0, 'analytic': False}
theory.sigma_sA = {'name': 'sigma_sA', 'value': 3.5, 'vary': True, 'fiducial': 3.5,
↪'prior_name': 'uniform', 'lower': 2.0, 'upper': 8.0, 'analytic': False}
theory.sigma_sB = {'name': 'sigma_sB', 'value': 5.0, 'vary': False, 'fiducial': 5.0,
↪'prior_name': 'uniform', 'lower': 3.0, 'upper': 10.0, 'analytic': False}
theory.sigma_so = {'name': 'sigma_so', 'value': 4.0, 'vary': False, 'fiducial': 4.0,
↪'prior_name': 'uniform', 'lower': 1.0, 'upper': 7, 'analytic': False}
#-------------------------------------------------------------------------------

#-------------------------------------------------------------------------------
# theory params
#-------------------------------------------------------------------------------
theory.F_AP = {'name': 'F_AP', 'value': 1.0, 'vary': False, 'expr': 'alpha_par/alpha_
↪perp', 'analytic': False}
theory.N = {'name': 'N', 'value': 0.0, 'vary': False, 'fiducial': 0.0, 'prior_name':
↪'uniform', 'lower': -500.0, 'upper': 500.0, 'analytic': False}
theory.NcBs = {'name': 'NcBs', 'value': 40236.785434927464, 'vary': False, 'expr':
↪'f1h_cBs / (fcB*(1 - fs)*nbar)', 'fiducial': 45000.0, 'analytic': False}
theory.NsBsB = {'name': 'NsBsB', 'value': 128534.1756949072, 'vary': False, 'expr':
↪'f1h_sBsB / (fsB**2 * fs**2 * nbar) * (fcB*(1 - fs) - fs*(1-fsB))', 'fiducial':␣
↪94500.0, 'analytic': False}
theory.Nsat_mult = {'name': 'Nsat_mult', 'value': 2.4, 'vary': True, 'min': 2.0,
↪'fiducial': 2.4, 'prior_name': 'normal', 'mu': 2.4, 'sigma': 0.2, 'analytic': False}
theory.alpha = {'name': 'alpha', 'value': 1.0, 'vary': False, 'expr': '(alpha_perp**2␣
↪* alpha_par)**(1./3)', 'analytic': False}
theory.alpha_drag = {'name': 'alpha_drag', 'value': 1.0, 'vary': False, 'fiducial': 1.
↪0, 'analytic': False}
```

(continues on next page)

(continued from previous page)

```
theory.alpha_par = {'name': 'alpha_par', 'value': 1.0, 'vary': True, 'fiducial': 1.0,
↪'prior_name': 'uniform', 'lower': 0.8, 'upper': 1.2, 'analytic': False}
theory.alpha_perp = {'name': 'alpha_perp', 'value': 1.0, 'vary': True, 'fiducial': 1.
↪0, 'prior_name': 'uniform', 'lower': 0.8, 'upper': 1.2, 'analytic': False}
theory.b1 = {'name': 'b1', 'value': 2.124276, 'vary': False, 'expr': '(1 - fs)*b1_c +␣
↪fs*b1_s', 'analytic': False}
theory.b1_c = {'name': 'b1_c', 'value': 1.9981383928571428, 'vary': False, 'expr':
↪'(1 - fcB)*b1_cA + fcB*b1_cB', 'analytic': False}
theory.b1_cA = {'name': 'b1_cA', 'value': 1.9, 'vary': True, 'fiducial': 1.9, 'prior_
↪name': 'uniform', 'lower': 1.2, 'upper': 2.5, 'analytic': False}
theory.b1_cB = {'name': 'b1_cB', 'value': 3.0028260869565218, 'vary': False, 'expr':
↪'(1-fsB)/(1+fsB*(1./Nsat_mult - 1)) * b1_sA +  (1 - (1-fsB)/(1+fsB*(1./Nsat_mult -␣
↪1))) * b1_sB', 'fiducial': 2.84, 'analytic': False}
theory.b1_s = {'name': 'b1_s', 'value': 3.2109999999999994, 'vary': False, 'expr':
↪'(1 - fsB)*b1_sA + fsB*b1_sB', 'analytic': False}
theory.b1_sA = {'name': 'b1_sA', 'value': 2.755, 'vary': False, 'expr': 'gamma_
↪b1sA*b1_cA', 'fiducial': 2.63, 'analytic': False}
theory.b1_sB = {'name': 'b1_sB', 'value': 3.8949999999999996, 'vary': False, 'expr':
↪'gamma_b1sB*b1_cA', 'fiducial': 3.62, 'analytic': False}
theory.b1sigma8 = {'name': 'b1sigma8', 'value': 1.29580836, 'vary': False, 'expr':
↪'b1*sigma8_z', 'analytic': False}
theory.epsilon = {'name': 'epsilon', 'value': 0.0, 'vary': False, 'expr': '(alpha_
↪perp/alpha_par)**(-1./3) - 1.0', 'analytic': False}
theory.f = {'name': 'f', 'value': 0.78, 'vary': True, 'fiducial': 0.78, 'prior_name':
↪'uniform', 'lower': 0.6, 'upper': 1.0, 'analytic': False}
theory.f1h_cBs = {'name': 'f1h_cBs', 'value': 1.0, 'vary': False, 'min': 0, 'fiducial
↪': 1.0, 'prior_name': 'normal', 'mu': 1.0, 'sigma': 0.75, 'analytic': False}
theory.f1h_sBsB = {'name': 'f1h_sBsB', 'value': 4.0, 'vary': True, 'min': 0.0,
↪'fiducial': 4.0, 'prior_name': 'normal', 'mu': 4.0, 'sigma': 1.0, 'analytic': False}
theory.f_so = {'name': 'f_so', 'value': 0.03, 'vary': False, 'fiducial': 0.03, 'prior_
↪name': 'normal', 'mu': 0.04, 'sigma': 0.02, 'analytic': False}
theory.fcB = {'name': 'fcB', 'value': 0.08898809523809524, 'vary': False, 'min': 0,
↪'max': 1, 'expr': 'fs / (1 - fs) * (1 + fsB*(1./Nsat_mult - 1))', 'fiducial': 0.089,
↪ 'analytic': False}
theory.fs = {'name': 'fs', 'value': 0.104, 'vary': True, 'min': 0.0, 'max': 1.0,
↪'fiducial': 0.104, 'prior_name': 'uniform', 'lower': 0.0, 'upper': 0.25, 'analytic
↪': False}
theory.fsB = {'name': 'fsB', 'value': 0.4, 'vary': True, 'min': 0.0, 'max': 1,
↪'fiducial': 0.4, 'prior_name': 'uniform', 'lower': 0.0, 'upper': 1.0, 'analytic':␣
↪False}
theory.fsigma8 = {'name': 'fsigma8', 'value': 0.4758, 'vary': False, 'expr':
↪'f*sigma8_z', 'analytic': False}
theory.gamma_b1cB = {'name': 'gamma_b1cB', 'value': 0.4, 'vary': False, 'min': 0.0,
↪'max': 1.0, 'fiducial': 0.4, 'prior_name': 'normal', 'mu': 0.4, 'sigma': 0.2,
↪'analytic': False}
theory.gamma_b1sA = {'name': 'gamma_b1sA', 'value': 1.45, 'vary': True, 'min': 1.0,
↪'fiducial': 1.45, 'prior_name': 'normal', 'mu': 1.45, 'sigma': 0.3, 'analytic':␣
↪False}
theory.gamma_b1sB = {'name': 'gamma_b1sB', 'value': 2.05, 'vary': True, 'min': 1.0,
↪'fiducial': 2.05, 'prior_name': 'normal', 'mu': 2.05, 'sigma': 0.3, 'analytic':␣
↪False}
theory.nbar = {'name': 'nbar', 'value': 0.0003117, 'vary': False, 'fiducial': 0.
↪0003117, 'analytic': False}
theory.sigma8_z = {'name': 'sigma8_z', 'value': 0.61, 'vary': True, 'fiducial': 0.61,
↪'prior_name': 'uniform', 'lower': 0.3, 'upper': 0.9, 'analytic': False}
theory.sigma_c = {'name': 'sigma_c', 'value': 1.0, 'vary': True, 'fiducial': 1.0,
↪'prior_name': 'uniform', 'lower': 0.0, 'upper': 3.0, 'analytic': False}
```

(continues on next page)

(continued from previous page)

```
theory.sigma_sA = {'name': 'sigma_sA', 'value': 3.5, 'vary': True, 'fiducial': 3.5,
↪'prior_name': 'uniform', 'lower': 2.0, 'upper': 8.0, 'analytic': False}
theory.sigma_sB = {'name': 'sigma_sB', 'value': 5.0, 'vary': False, 'fiducial': 5.0,
↪'prior_name': 'uniform', 'lower': 3.0, 'upper': 10.0, 'analytic': False}
theory.sigma_so = {'name': 'sigma_so', 'value': 4.0, 'vary': False, 'fiducial': 4.0,
↪'prior_name': 'uniform', 'lower': 1.0, 'upper': 7, 'analytic': False}
#-------------------------------------------------------------------------------

#-------------------------------------------------------------------------------
# theory params
#-------------------------------------------------------------------------------
theory.F_AP = {'name': 'F_AP', 'value': 1.0, 'vary': False, 'expr': 'alpha_par/alpha_
↪perp', 'analytic': False}
theory.N = {'name': 'N', 'value': 0.0, 'vary': False, 'fiducial': 0.0, 'prior_name':
↪'uniform', 'lower': -500.0, 'upper': 500.0, 'analytic': False}
theory.NcBs = {'name': 'NcBs', 'value': 40236.785434927464, 'vary': False, 'expr':
↪'f1h_cBs / (fcB*(1 - fs)*nbar)', 'fiducial': 45000.0, 'analytic': False}
theory.NsBsB = {'name': 'NsBsB', 'value': 128534.1756949072, 'vary': False, 'expr':
↪'f1h_sBsB / (fsB**2 * fs**2 * nbar) * (fcB*(1 - fs) - fs*(1-fsB))', 'fiducial':␣
↪94500.0, 'analytic': False}
theory.Nsat_mult = {'name': 'Nsat_mult', 'value': 2.4, 'vary': True, 'min': 2.0,
↪'fiducial': 2.4, 'prior_name': 'normal', 'mu': 2.4, 'sigma': 0.2, 'analytic': False}
theory.alpha = {'name': 'alpha', 'value': 1.0, 'vary': False, 'expr': '(alpha_perp**2␣
↪* alpha_par)**(1./3)', 'analytic': False}
theory.alpha_drag = {'name': 'alpha_drag', 'value': 1.0, 'vary': False, 'fiducial': 1.
↪0, 'analytic': False}
theory.alpha_par = {'name': 'alpha_par', 'value': 1.0, 'vary': True, 'fiducial': 1.0,
↪'prior_name': 'uniform', 'lower': 0.8, 'upper': 1.2, 'analytic': False}
theory.alpha_perp = {'name': 'alpha_perp', 'value': 1.0, 'vary': True, 'fiducial': 1.
↪0, 'prior_name': 'uniform', 'lower': 0.8, 'upper': 1.2, 'analytic': False}
theory.b1 = {'name': 'b1', 'value': 2.124276, 'vary': False, 'expr': '(1 - fs)*b1_c +␣
↪fs*b1_s', 'analytic': False}
theory.b1_c = {'name': 'b1_c', 'value': 1.9981383928571428, 'vary': False, 'expr':
↪'(1 - fcB)*b1_cA + fcB*b1_cB', 'analytic': False}
theory.b1_cA = {'name': 'b1_cA', 'value': 1.9, 'vary': True, 'fiducial': 1.9, 'prior_
↪name': 'uniform', 'lower': 1.2, 'upper': 2.5, 'analytic': False}
theory.b1_cB = {'name': 'b1_cB', 'value': 3.0028260869565218, 'vary': False, 'expr':
↪'(1-fsB)/(1+fsB*(1./Nsat_mult - 1)) * b1_sA +  (1 - (1-fsB)/(1+fsB*(1./Nsat_mult -␣
↪1))) * b1_sB', 'fiducial': 2.84, 'analytic': False}
theory.b1_s = {'name': 'b1_s', 'value': 3.2109999999999994, 'vary': False, 'expr':
↪'(1 - fsB)*b1_sA + fsB*b1_sB', 'analytic': False}
theory.b1_sA = {'name': 'b1_sA', 'value': 2.755, 'vary': False, 'expr': 'gamma_
↪b1sA*b1_cA', 'fiducial': 2.63, 'analytic': False}
theory.b1_sB = {'name': 'b1_sB', 'value': 3.8949999999999996, 'vary': False, 'expr':
↪'gamma_b1sB*b1_cA', 'fiducial': 3.62, 'analytic': False}
theory.b1sigma8 = {'name': 'b1sigma8', 'value': 1.29580836, 'vary': False, 'expr':
↪'b1*sigma8_z', 'analytic': False}
theory.epsilon = {'name': 'epsilon', 'value': 0.0, 'vary': False, 'expr': '(alpha_
↪perp/alpha_par)**(-1./3) - 1.0', 'analytic': False}
theory.f = {'name': 'f', 'value': 0.78, 'vary': True, 'fiducial': 0.78, 'prior_name':
↪'uniform', 'lower': 0.6, 'upper': 1.0, 'analytic': False}
theory.f1h_cBs = {'name': 'f1h_cBs', 'value': 1.0, 'vary': False, 'min': 0, 'fiducial
↪': 1.0, 'prior_name': 'normal', 'mu': 1.0, 'sigma': 0.75, 'analytic': False}
theory.f1h_sBsB = {'name': 'f1h_sBsB', 'value': 4.0, 'vary': True, 'min': 0.0,
↪'fiducial': 4.0, 'prior_name': 'normal', 'mu': 4.0, 'sigma': 1.0, 'analytic': False}
theory.f_so = {'name': 'f_so', 'value': 0.03, 'vary': False, 'fiducial': 0.03, 'prior_
↪name': 'normal', 'mu': 0.04, 'sigma': 0.02, 'analytic': False}
```

(continues on next page)

(continued from previous page)

```
theory.fcB = {'name': 'fcB', 'value': 0.08898809523809524, 'vary': False, 'min': 0,
↪'max': 1, 'expr': 'fs / (1 - fs) * (1 + fsB*(1./Nsat_mult - 1))', 'fiducial': 0.089,
↪ 'analytic': False}
theory.fs = {'name': 'fs', 'value': 0.104, 'vary': True, 'min': 0.0, 'max': 1.0,
↪'fiducial': 0.104, 'prior_name': 'uniform', 'lower': 0.0, 'upper': 0.25, 'analytic
↪': False}
theory.fsB = {'name': 'fsB', 'value': 0.4, 'vary': True, 'min': 0.0, 'max': 1,
↪'fiducial': 0.4, 'prior_name': 'uniform', 'lower': 0.0, 'upper': 1.0, 'analytic':␣
↪False}
theory.fsigma8 = {'name': 'fsigma8', 'value': 0.4758, 'vary': False, 'expr':
↪'f*sigma8_z', 'analytic': False}
theory.gamma_b1cB = {'name': 'gamma_b1cB', 'value': 0.4, 'vary': False, 'min': 0.0,
↪'max': 1.0, 'fiducial': 0.4, 'prior_name': 'normal', 'mu': 0.4, 'sigma': 0.2,
↪'analytic': False}
theory.gamma_b1sA = {'name': 'gamma_b1sA', 'value': 1.45, 'vary': True, 'min': 1.0,
↪'fiducial': 1.45, 'prior_name': 'normal', 'mu': 1.45, 'sigma': 0.3, 'analytic':␣
↪False}
theory.gamma_b1sB = {'name': 'gamma_b1sB', 'value': 2.05, 'vary': True, 'min': 1.0,
↪'fiducial': 2.05, 'prior_name': 'normal', 'mu': 2.05, 'sigma': 0.3, 'analytic':␣
↪False}
theory.nbar = {'name': 'nbar', 'value': 0.0003117, 'vary': False, 'fiducial': 0.
↪0003117, 'analytic': False}
theory.sigma8_z = {'name': 'sigma8_z', 'value': 0.61, 'vary': True, 'fiducial': 0.61,
↪'prior_name': 'uniform', 'lower': 0.3, 'upper': 0.9, 'analytic': False}
theory.sigma_c = {'name': 'sigma_c', 'value': 1.0, 'vary': True, 'fiducial': 1.0,
↪'prior_name': 'uniform', 'lower': 0.0, 'upper': 3.0, 'analytic': False}
theory.sigma_sA = {'name': 'sigma_sA', 'value': 3.5, 'vary': True, 'fiducial': 3.5,
↪'prior_name': 'uniform', 'lower': 2.0, 'upper': 8.0, 'analytic': False}
theory.sigma_sB = {'name': 'sigma_sB', 'value': 5.0, 'vary': False, 'fiducial': 5.0,
↪'prior_name': 'uniform', 'lower': 3.0, 'upper': 10.0, 'analytic': False}
theory.sigma_so = {'name': 'sigma_so', 'value': 4.0, 'vary': False, 'fiducial': 4.0,
↪'prior_name': 'uniform', 'lower': 1.0, 'upper': 7, 'analytic': False}
#------------------------------------------------------------------------------

#------------------------------------------------------------------------------
# theory params
#------------------------------------------------------------------------------
theory.F_AP = {'name': 'F_AP', 'value': 1.0, 'vary': False, 'expr': 'alpha_par/alpha_
↪perp', 'analytic': False}
theory.N = {'name': 'N', 'value': 0.0, 'vary': False, 'fiducial': 0.0, 'prior_name':
↪'uniform', 'lower': -500.0, 'upper': 500.0, 'analytic': False}
theory.NcBs = {'name': 'NcBs', 'value': 40236.785434927464, 'vary': False, 'expr':
↪'f1h_cBs / (fcB*(1 - fs)*nbar)', 'fiducial': 45000.0, 'analytic': False}
theory.NsBsB = {'name': 'NsBsB', 'value': 128534.1756949072, 'vary': False, 'expr':
↪'f1h_sBsB / (fsB**2 * fs**2 * nbar) * (fcB*(1 - fs) - fs*(1-fsB))', 'fiducial':␣
↪94500.0, 'analytic': False}
theory.Nsat_mult = {'name': 'Nsat_mult', 'value': 2.4, 'vary': True, 'min': 2.0,
↪'fiducial': 2.4, 'prior_name': 'normal', 'mu': 2.4, 'sigma': 0.2, 'analytic': False}
theory.alpha = {'name': 'alpha', 'value': 1.0, 'vary': False, 'expr': '(alpha_perp**2␣
↪* alpha_par)**(1./3)', 'analytic': False}
theory.alpha_drag = {'name': 'alpha_drag', 'value': 1.0, 'vary': False, 'fiducial': 1.
↪0, 'analytic': False}
theory.alpha_par = {'name': 'alpha_par', 'value': 1.0, 'vary': True, 'fiducial': 1.0,
↪'prior_name': 'uniform', 'lower': 0.8, 'upper': 1.2, 'analytic': False}
theory.alpha_perp = {'name': 'alpha_perp', 'value': 1.0, 'vary': True, 'fiducial': 1.
↪0, 'prior_name': 'uniform', 'lower': 0.8, 'upper': 1.2, 'analytic': False}
theory.b1 = {'name': 'b1', 'value': 2.124276, 'vary': False, 'expr': '(1 - fs)*b1_c +␣
↪fs*b1_s', 'analytic': False}
```

(continues on next page)

(continued from previous page)

```
theory.b1_c = {'name': 'b1_c', 'value': 1.9981383928571428, 'vary': False, 'expr':
↪'(1 - fcB)*b1_cA + fcB*b1_cB', 'analytic': False}
theory.b1_cA = {'name': 'b1_cA', 'value': 1.9, 'vary': True, 'fiducial': 1.9, 'prior_
↪name': 'uniform', 'lower': 1.2, 'upper': 2.5, 'analytic': False}
theory.b1_cB = {'name': 'b1_cB', 'value': 3.0028260869565218, 'vary': False, 'expr':
↪'(1-fsB)/(1+fsB*(1./Nsat_mult - 1)) * b1_sA +  (1 - (1-fsB)/(1+fsB*(1./Nsat_mult -␣
↪1))) * b1_sB', 'fiducial': 2.84, 'analytic': False}
theory.b1_s = {'name': 'b1_s', 'value': 3.2109999999999994, 'vary': False, 'expr':
↪'(1 - fsB)*b1_sA + fsB*b1_sB', 'analytic': False}
theory.b1_sA = {'name': 'b1_sA', 'value': 2.755, 'vary': False, 'expr': 'gamma_
↪b1sA*b1_cA', 'fiducial': 2.63, 'analytic': False}
theory.b1_sB = {'name': 'b1_sB', 'value': 3.8949999999999996, 'vary': False, 'expr':
↪'gamma_b1sB*b1_cA', 'fiducial': 3.62, 'analytic': False}
theory.b1sigma8 = {'name': 'b1sigma8', 'value': 1.29580836, 'vary': False, 'expr':
↪'b1*sigma8_z', 'analytic': False}
theory.epsilon = {'name': 'epsilon', 'value': 0.0, 'vary': False, 'expr': '(alpha_
↪perp/alpha_par)**(-1./3) - 1.0', 'analytic': False}
theory.f = {'name': 'f', 'value': 0.78, 'vary': True, 'fiducial': 0.78, 'prior_name':
↪'uniform', 'lower': 0.6, 'upper': 1.0, 'analytic': False}
theory.f1h_cBs = {'name': 'f1h_cBs', 'value': 1.0, 'vary': False, 'min': 0, 'fiducial
↪': 1.0, 'prior_name': 'normal', 'mu': 1.0, 'sigma': 0.75, 'analytic': False}
theory.f1h_sBsB = {'name': 'f1h_sBsB', 'value': 4.0, 'vary': True, 'min': 0.0,
↪'fiducial': 4.0, 'prior_name': 'normal', 'mu': 4.0, 'sigma': 1.0, 'analytic': False}
theory.f_so = {'name': 'f_so', 'value': 0.03, 'vary': False, 'fiducial': 0.03, 'prior_
↪name': 'normal', 'mu': 0.04, 'sigma': 0.02, 'analytic': False}
theory.fcB = {'name': 'fcB', 'value': 0.08898809523809524, 'vary': False, 'min': 0,
↪'max': 1, 'expr': 'fs / (1 - fs) * (1 + fsB*(1./Nsat_mult - 1))', 'fiducial': 0.089,
↪ 'analytic': False}
theory.fs = {'name': 'fs', 'value': 0.104, 'vary': True, 'min': 0.0, 'max': 1.0,
↪'fiducial': 0.104, 'prior_name': 'uniform', 'lower': 0.0, 'upper': 0.25, 'analytic
↪': False}
theory.fsB = {'name': 'fsB', 'value': 0.4, 'vary': True, 'min': 0.0, 'max': 1,
↪'fiducial': 0.4, 'prior_name': 'uniform', 'lower': 0.0, 'upper': 1.0, 'analytic':␣
↪False}
theory.fsigma8 = {'name': 'fsigma8', 'value': 0.4758, 'vary': False, 'expr':
↪'f*sigma8_z', 'analytic': False}
theory.gamma_b1cB = {'name': 'gamma_b1cB', 'value': 0.4, 'vary': False, 'min': 0.0,
↪'max': 1.0, 'fiducial': 0.4, 'prior_name': 'normal', 'mu': 0.4, 'sigma': 0.2,
↪'analytic': False}
theory.gamma_b1sA = {'name': 'gamma_b1sA', 'value': 1.45, 'vary': True, 'min': 1.0,
↪'fiducial': 1.45, 'prior_name': 'normal', 'mu': 1.45, 'sigma': 0.3, 'analytic':␣
↪False}
theory.gamma_b1sB = {'name': 'gamma_b1sB', 'value': 2.05, 'vary': True, 'min': 1.0,
↪'fiducial': 2.05, 'prior_name': 'normal', 'mu': 2.05, 'sigma': 0.3, 'analytic':␣
↪False}
theory.nbar = {'name': 'nbar', 'value': 0.0003117, 'vary': False, 'fiducial': 0.
↪0003117, 'analytic': False}
theory.sigma8_z = {'name': 'sigma8_z', 'value': 0.61, 'vary': True, 'fiducial': 0.61,
↪'prior_name': 'uniform', 'lower': 0.3, 'upper': 0.9, 'analytic': False}
theory.sigma_c = {'name': 'sigma_c', 'value': 1.0, 'vary': True, 'fiducial': 1.0,
↪'prior_name': 'uniform', 'lower': 0.0, 'upper': 3.0, 'analytic': False}
theory.sigma_sA = {'name': 'sigma_sA', 'value': 3.5, 'vary': True, 'fiducial': 3.5,
↪'prior_name': 'uniform', 'lower': 2.0, 'upper': 8.0, 'analytic': False}
theory.sigma_sB = {'name': 'sigma_sB', 'value': 5.0, 'vary': False, 'fiducial': 5.0,
↪'prior_name': 'uniform', 'lower': 3.0, 'upper': 10.0, 'analytic': False}
theory.sigma_so = {'name': 'sigma_so', 'value': 4.0, 'vary': False, 'fiducial': 4.0,
↪'prior_name': 'uniform', 'lower': 1.0, 'upper': 7, 'analytic': False}
```

(continues on next page)

(continued from previous page)

```
#-------------------------------------------------------------------------------

#-------------------------------------------------------------------------------
# theory params
#-------------------------------------------------------------------------------
theory.F_AP = {'name': 'F_AP', 'value': 1.0, 'vary': False, 'expr': 'alpha_par/alpha_
↪perp', 'analytic': False}
theory.N = {'name': 'N', 'value': 0.0, 'vary': False, 'fiducial': 0.0, 'prior_name':
↪'uniform', 'lower': -500.0, 'upper': 500.0, 'analytic': False}
theory.NcBs = {'name': 'NcBs', 'value': 40236.785434927464, 'vary': False, 'expr':
↪'f1h_cBs / (fcB*(1 - fs)*nbar)', 'fiducial': 45000.0, 'analytic': False}
theory.NsBsB = {'name': 'NsBsB', 'value': 128534.1756949072, 'vary': False, 'expr':
↪'f1h_sBsB / (fsB**2 * fs**2 * nbar) * (fcB*(1 - fs) - fs*(1-fsB))', 'fiducial':␣
↪94500.0, 'analytic': False}
theory.Nsat_mult = {'name': 'Nsat_mult', 'value': 2.4, 'vary': True, 'min': 2.0,
↪'fiducial': 2.4, 'prior_name': 'normal', 'mu': 2.4, 'sigma': 0.2, 'analytic': False}
theory.alpha = {'name': 'alpha', 'value': 1.0, 'vary': False, 'expr': '(alpha_perp**2␣
↪* alpha_par)**(1./3)', 'analytic': False}
theory.alpha_drag = {'name': 'alpha_drag', 'value': 1.0, 'vary': False, 'fiducial': 1.
↪0, 'analytic': False}
theory.alpha_par = {'name': 'alpha_par', 'value': 1.0, 'vary': True, 'fiducial': 1.0,
↪'prior_name': 'uniform', 'lower': 0.8, 'upper': 1.2, 'analytic': False}
theory.alpha_perp = {'name': 'alpha_perp', 'value': 1.0, 'vary': True, 'fiducial': 1.
↪0, 'prior_name': 'uniform', 'lower': 0.8, 'upper': 1.2, 'analytic': False}
theory.b1 = {'name': 'b1', 'value': 2.124276, 'vary': False, 'expr': '(1 - fs)*b1_c +␣
↪fs*b1_s', 'analytic': False}
theory.b1_c = {'name': 'b1_c', 'value': 1.9981383928571428, 'vary': False, 'expr':
↪'(1 - fcB)*b1_cA + fcB*b1_cB', 'analytic': False}
theory.b1_cA = {'name': 'b1_cA', 'value': 1.9, 'vary': True, 'fiducial': 1.9, 'prior_
↪name': 'uniform', 'lower': 1.2, 'upper': 2.5, 'analytic': False}
theory.b1_cB = {'name': 'b1_cB', 'value': 3.0028260869565218, 'vary': False, 'expr':
↪'(1-fsB)/(1+fsB*(1./Nsat_mult - 1)) * b1_sA +  (1 - (1-fsB)/(1+fsB*(1./Nsat_mult -␣
↪1))) * b1_sB', 'fiducial': 2.84, 'analytic': False}
theory.b1_s = {'name': 'b1_s', 'value': 3.2109999999999994, 'vary': False, 'expr':
↪'(1 - fsB)*b1_sA + fsB*b1_sB', 'analytic': False}
theory.b1_sA = {'name': 'b1_sA', 'value': 2.755, 'vary': False, 'expr': 'gamma_
↪b1sA*b1_cA', 'fiducial': 2.63, 'analytic': False}
theory.b1_sB = {'name': 'b1_sB', 'value': 3.8949999999999996, 'vary': False, 'expr':
↪'gamma_b1sB*b1_cA', 'fiducial': 3.62, 'analytic': False}
theory.b1sigma8 = {'name': 'b1sigma8', 'value': 1.29580836, 'vary': False, 'expr':
↪'b1*sigma8_z', 'analytic': False}
theory.epsilon = {'name': 'epsilon', 'value': 0.0, 'vary': False, 'expr': '(alpha_
↪perp/alpha_par)**(-1./3) - 1.0', 'analytic': False}
theory.f = {'name': 'f', 'value': 0.78, 'vary': True, 'fiducial': 0.78, 'prior_name':
↪'uniform', 'lower': 0.6, 'upper': 1.0, 'analytic': False}
theory.f1h_cBs = {'name': 'f1h_cBs', 'value': 1.0, 'vary': False, 'min': 0, 'fiducial
↪': 1.0, 'prior_name': 'normal', 'mu': 1.0, 'sigma': 0.75, 'analytic': False}
theory.f1h_sBsB = {'name': 'f1h_sBsB', 'value': 4.0, 'vary': True, 'min': 0.0,
↪'fiducial': 4.0, 'prior_name': 'normal', 'mu': 4.0, 'sigma': 1.0, 'analytic': False}
theory.f_so = {'name': 'f_so', 'value': 0.03, 'vary': False, 'fiducial': 0.03, 'prior_
↪name': 'normal', 'mu': 0.04, 'sigma': 0.02, 'analytic': False}
theory.fcB = {'name': 'fcB', 'value': 0.08898809523809524, 'vary': False, 'min': 0,
↪'max': 1, 'expr': 'fs / (1 - fs) * (1 + fsB*(1./Nsat_mult - 1))', 'fiducial': 0.089,
↪ 'analytic': False}
theory.fs = {'name': 'fs', 'value': 0.104, 'vary': True, 'min': 0.0, 'max': 1.0,
↪'fiducial': 0.104, 'prior_name': 'uniform', 'lower': 0.0, 'upper': 0.25, 'analytic
↪': False}
```

(continues on next page)

(continued from previous page)

```
theory.fsB = {'name': 'fsB', 'value': 0.4, 'vary': True, 'min': 0.0, 'max': 1,
↪'fiducial': 0.4, 'prior_name': 'uniform', 'lower': 0.0, 'upper': 1.0, 'analytic':␣
↪False}
theory.fsigma8 = {'name': 'fsigma8', 'value': 0.4758, 'vary': False, 'expr':
↪'f*sigma8_z', 'analytic': False}
theory.gamma_b1cB = {'name': 'gamma_b1cB', 'value': 0.4, 'vary': False, 'min': 0.0,
↪'max': 1.0, 'fiducial': 0.4, 'prior_name': 'normal', 'mu': 0.4, 'sigma': 0.2,
↪'analytic': False}
theory.gamma_b1sA = {'name': 'gamma_b1sA', 'value': 1.45, 'vary': True, 'min': 1.0,
↪'fiducial': 1.45, 'prior_name': 'normal', 'mu': 1.45, 'sigma': 0.3, 'analytic':␣
↪False}
theory.gamma_b1sB = {'name': 'gamma_b1sB', 'value': 2.05, 'vary': True, 'min': 1.0,
↪'fiducial': 2.05, 'prior_name': 'normal', 'mu': 2.05, 'sigma': 0.3, 'analytic':␣
↪False}
theory.nbar = {'name': 'nbar', 'value': 0.0003117, 'vary': False, 'fiducial': 0.
↪0003117, 'analytic': False}
theory.sigma8_z = {'name': 'sigma8_z', 'value': 0.61, 'vary': True, 'fiducial': 0.61,
↪'prior_name': 'uniform', 'lower': 0.3, 'upper': 0.9, 'analytic': False}
theory.sigma_c = {'name': 'sigma_c', 'value': 1.0, 'vary': True, 'fiducial': 1.0,
↪'prior_name': 'uniform', 'lower': 0.0, 'upper': 3.0, 'analytic': False}
theory.sigma_sA = {'name': 'sigma_sA', 'value': 3.5, 'vary': True, 'fiducial': 3.5,
↪'prior_name': 'uniform', 'lower': 2.0, 'upper': 8.0, 'analytic': False}
theory.sigma_sB = {'name': 'sigma_sB', 'value': 5.0, 'vary': False, 'fiducial': 5.0,
↪'prior_name': 'uniform', 'lower': 3.0, 'upper': 10.0, 'analytic': False}
theory.sigma_so = {'name': 'sigma_so', 'value': 4.0, 'vary': False, 'fiducial': 4.0,
↪'prior_name': 'uniform', 'lower': 1.0, 'upper': 7, 'analytic': False}
#------------------------------------------------------------------------------

#------------------------------------------------------------------------------
# theory params
#------------------------------------------------------------------------------
theory.F_AP = {'name': 'F_AP', 'value': 1.0, 'vary': False, 'expr': 'alpha_par/alpha_
↪perp', 'analytic': False}
theory.N = {'name': 'N', 'value': 0.0, 'vary': False, 'fiducial': 0.0, 'prior_name':
↪'uniform', 'lower': -500.0, 'upper': 500.0, 'analytic': False}
theory.NcBs = {'name': 'NcBs', 'value': 40236.785434927464, 'vary': False, 'expr':
↪'f1h_cBs / (fcB*(1 - fs)*nbar)', 'fiducial': 45000.0, 'analytic': False}
theory.NsBsB = {'name': 'NsBsB', 'value': 128534.1756949072, 'vary': False, 'expr':
↪'f1h_sBsB / (fsB**2 * fs**2 * nbar) * (fcB*(1 - fs) - fs*(1-fsB))', 'fiducial':␣
↪94500.0, 'analytic': False}
theory.Nsat_mult = {'name': 'Nsat_mult', 'value': 2.4, 'vary': True, 'min': 2.0,
↪'fiducial': 2.4, 'prior_name': 'normal', 'mu': 2.4, 'sigma': 0.2, 'analytic': False}
theory.alpha = {'name': 'alpha', 'value': 1.0, 'vary': False, 'expr': '(alpha_perp**2␣
↪* alpha_par)**(1./3)', 'analytic': False}
theory.alpha_drag = {'name': 'alpha_drag', 'value': 1.0, 'vary': False, 'fiducial': 1.
↪0, 'analytic': False}
theory.alpha_par = {'name': 'alpha_par', 'value': 1.0, 'vary': True, 'fiducial': 1.0,
↪'prior_name': 'uniform', 'lower': 0.8, 'upper': 1.2, 'analytic': False}
theory.alpha_perp = {'name': 'alpha_perp', 'value': 1.0, 'vary': True, 'fiducial': 1.
↪0, 'prior_name': 'uniform', 'lower': 0.8, 'upper': 1.2, 'analytic': False}
theory.b1 = {'name': 'b1', 'value': 2.124276, 'vary': False, 'expr': '(1 - fs)*b1_c +␣
↪fs*b1_s', 'analytic': False}
theory.b1_c = {'name': 'b1_c', 'value': 1.9981383928571428, 'vary': False, 'expr':
↪'(1 - fcB)*b1_cA + fcB*b1_cB', 'analytic': False}
theory.b1_cA = {'name': 'b1_cA', 'value': 1.9, 'vary': True, 'fiducial': 1.9, 'prior_
↪name': 'uniform', 'lower': 1.2, 'upper': 2.5, 'analytic': False}
theory.b1_cB = {'name': 'b1_cB', 'value': 3.0028260869565218, 'vary': False, 'expr':
↪'(1-fsB)/(1+fsB*(1./Nsat_mult - 1)) * b1_sA +  (1 - (1-fsB)/(1+fsB*(1./Nsat_mult -␣
↪1))) * b1_sB', 'fiducial': 2.84, 'analytic': False}
```

(continues on next page)

(continued from previous page)

```
theory.b1_s = {'name': 'b1_s', 'value': 3.2109999999999994, 'vary': False, 'expr':
↪'(1 - fsB)*b1_sA + fsB*b1_sB', 'analytic': False}
theory.b1_sA = {'name': 'b1_sA', 'value': 2.755, 'vary': False, 'expr': 'gamma_
↪b1sA*b1_cA', 'fiducial': 2.63, 'analytic': False}
theory.b1_sB = {'name': 'b1_sB', 'value': 3.8949999999999996, 'vary': False, 'expr':
↪'gamma_b1sB*b1_cA', 'fiducial': 3.62, 'analytic': False}
theory.b1sigma8 = {'name': 'b1sigma8', 'value': 1.29580836, 'vary': False, 'expr':
↪'b1*sigma8_z', 'analytic': False}
theory.epsilon = {'name': 'epsilon', 'value': 0.0, 'vary': False, 'expr': '(alpha_
↪perp/alpha_par)**(-1./3) - 1.0', 'analytic': False}
theory.f = {'name': 'f', 'value': 0.78, 'vary': True, 'fiducial': 0.78, 'prior_name':
↪'uniform', 'lower': 0.6, 'upper': 1.0, 'analytic': False}
theory.f1h_cBs = {'name': 'f1h_cBs', 'value': 1.0, 'vary': False, 'min': 0, 'fiducial
↪': 1.0, 'prior_name': 'normal', 'mu': 1.0, 'sigma': 0.75, 'analytic': False}
theory.f1h_sBsB = {'name': 'f1h_sBsB', 'value': 4.0, 'vary': True, 'min': 0.0,
↪'fiducial': 4.0, 'prior_name': 'normal', 'mu': 4.0, 'sigma': 1.0, 'analytic': False}
theory.f_so = {'name': 'f_so', 'value': 0.03, 'vary': False, 'fiducial': 0.03, 'prior_
↪name': 'normal', 'mu': 0.04, 'sigma': 0.02, 'analytic': False}
theory.fcB = {'name': 'fcB', 'value': 0.08898809523809524, 'vary': False, 'min': 0,
↪'max': 1, 'expr': 'fs / (1 - fs) * (1 + fsB*(1./Nsat_mult - 1))', 'fiducial': 0.089,
↪ 'analytic': False}
theory.fs = {'name': 'fs', 'value': 0.104, 'vary': True, 'min': 0.0, 'max': 1.0,
↪'fiducial': 0.104, 'prior_name': 'uniform', 'lower': 0.0, 'upper': 0.25, 'analytic
↪': False}
theory.fsB = {'name': 'fsB', 'value': 0.4, 'vary': True, 'min': 0.0, 'max': 1,
↪'fiducial': 0.4, 'prior_name': 'uniform', 'lower': 0.0, 'upper': 1.0, 'analytic':␣
↪False}
theory.fsigma8 = {'name': 'fsigma8', 'value': 0.4758, 'vary': False, 'expr':
↪'f*sigma8_z', 'analytic': False}
theory.gamma_b1cB = {'name': 'gamma_b1cB', 'value': 0.4, 'vary': False, 'min': 0.0,
↪'max': 1.0, 'fiducial': 0.4, 'prior_name': 'normal', 'mu': 0.4, 'sigma': 0.2,
↪'analytic': False}
theory.gamma_b1sA = {'name': 'gamma_b1sA', 'value': 1.45, 'vary': True, 'min': 1.0,
↪'fiducial': 1.45, 'prior_name': 'normal', 'mu': 1.45, 'sigma': 0.3, 'analytic':␣
↪False}
theory.gamma_b1sB = {'name': 'gamma_b1sB', 'value': 2.05, 'vary': True, 'min': 1.0,
↪'fiducial': 2.05, 'prior_name': 'normal', 'mu': 2.05, 'sigma': 0.3, 'analytic':␣
↪False}
theory.nbar = {'name': 'nbar', 'value': 0.0003117, 'vary': False, 'fiducial': 0.
↪0003117, 'analytic': False}
theory.sigma8_z = {'name': 'sigma8_z', 'value': 0.61, 'vary': True, 'fiducial': 0.61,
↪'prior_name': 'uniform', 'lower': 0.3, 'upper': 0.9, 'analytic': False}
theory.sigma_c = {'name': 'sigma_c', 'value': 1.0, 'vary': True, 'fiducial': 1.0,
↪'prior_name': 'uniform', 'lower': 0.0, 'upper': 3.0, 'analytic': False}
theory.sigma_sA = {'name': 'sigma_sA', 'value': 3.5, 'vary': True, 'fiducial': 3.5,
↪'prior_name': 'uniform', 'lower': 2.0, 'upper': 8.0, 'analytic': False}
theory.sigma_sB = {'name': 'sigma_sB', 'value': 5.0, 'vary': False, 'fiducial': 5.0,
↪'prior_name': 'uniform', 'lower': 3.0, 'upper': 10.0, 'analytic': False}
theory.sigma_so = {'name': 'sigma_so', 'value': 4.0, 'vary': False, 'fiducial': 4.0,
↪'prior_name': 'uniform', 'lower': 1.0, 'upper': 7, 'analytic': False}
#------------------------------------------------------------------------------

#------------------------------------------------------------------------------
# theory params
#------------------------------------------------------------------------------
theory.F_AP = {'name': 'F_AP', 'value': 1.0, 'vary': False, 'expr': 'alpha_par/alpha_
↪perp', 'analytic': False}
```

(continues on next page)

(continued from previous page)

```
theory.N = {'name': 'N', 'value': 0.0, 'vary': False, 'fiducial': 0.0, 'prior_name':
↪'uniform', 'lower': -500.0, 'upper': 500.0, 'analytic': False}
theory.NcBs = {'name': 'NcBs', 'value': 40236.785434927464, 'vary': False, 'expr':
↪'f1h_cBs / (fcB*(1 - fs)*nbar)', 'fiducial': 45000.0, 'analytic': False}
theory.NsBsB = {'name': 'NsBsB', 'value': 128534.1756949072, 'vary': False, 'expr':
↪'f1h_sBsB / (fsB**2 * fs**2 * nbar) * (fcB*(1 - fs) - fs*(1-fsB))', 'fiducial':␣
↪94500.0, 'analytic': False}
theory.Nsat_mult = {'name': 'Nsat_mult', 'value': 2.4, 'vary': True, 'min': 2.0,
↪'fiducial': 2.4, 'prior_name': 'normal', 'mu': 2.4, 'sigma': 0.2, 'analytic': False}
theory.alpha = {'name': 'alpha', 'value': 1.0, 'vary': False, 'expr': '(alpha_perp**2␣
↪* alpha_par)**(1./3)', 'analytic': False}
theory.alpha_drag = {'name': 'alpha_drag', 'value': 1.0, 'vary': False, 'fiducial': 1.
↪0, 'analytic': False}
theory.alpha_par = {'name': 'alpha_par', 'value': 1.0, 'vary': True, 'fiducial': 1.0,
↪'prior_name': 'uniform', 'lower': 0.8, 'upper': 1.2, 'analytic': False}
theory.alpha_perp = {'name': 'alpha_perp', 'value': 1.0, 'vary': True, 'fiducial': 1.
↪0, 'prior_name': 'uniform', 'lower': 0.8, 'upper': 1.2, 'analytic': False}
theory.b1 = {'name': 'b1', 'value': 2.124276, 'vary': False, 'expr': '(1 - fs)*b1_c +␣
↪fs*b1_s', 'analytic': False}
theory.b1_c = {'name': 'b1_c', 'value': 1.9981383928571428, 'vary': False, 'expr':
↪'(1 - fcB)*b1_cA + fcB*b1_cB', 'analytic': False}
theory.b1_cA = {'name': 'b1_cA', 'value': 1.9, 'vary': True, 'fiducial': 1.9, 'prior_
↪name': 'uniform', 'lower': 1.2, 'upper': 2.5, 'analytic': False}
theory.b1_cB = {'name': 'b1_cB', 'value': 3.0028260869565218, 'vary': False, 'expr':
↪'(1-fsB)/(1+fsB*(1./Nsat_mult - 1)) * b1_sA +  (1 - (1-fsB)/(1+fsB*(1./Nsat_mult -␣
↪1))) * b1_sB', 'fiducial': 2.84, 'analytic': False}
theory.b1_s = {'name': 'b1_s', 'value': 3.2109999999999994, 'vary': False, 'expr':
↪'(1 - fsB)*b1_sA + fsB*b1_sB', 'analytic': False}
theory.b1_sA = {'name': 'b1_sA', 'value': 2.755, 'vary': False, 'expr': 'gamma_
↪b1sA*b1_cA', 'fiducial': 2.63, 'analytic': False}
theory.b1_sB = {'name': 'b1_sB', 'value': 3.8949999999999996, 'vary': False, 'expr':
↪'gamma_b1sB*b1_cA', 'fiducial': 3.62, 'analytic': False}
theory.b1sigma8 = {'name': 'b1sigma8', 'value': 1.29580836, 'vary': False, 'expr':
↪'b1*sigma8_z', 'analytic': False}
theory.epsilon = {'name': 'epsilon', 'value': 0.0, 'vary': False, 'expr': '(alpha_
↪perp/alpha_par)**(-1./3) - 1.0', 'analytic': False}
theory.f = {'name': 'f', 'value': 0.78, 'vary': True, 'fiducial': 0.78, 'prior_name':
↪'uniform', 'lower': 0.6, 'upper': 1.0, 'analytic': False}
theory.f1h_cBs = {'name': 'f1h_cBs', 'value': 1.0, 'vary': False, 'min': 0, 'fiducial
↪': 1.0, 'prior_name': 'normal', 'mu': 1.0, 'sigma': 0.75, 'analytic': False}
theory.f1h_sBsB = {'name': 'f1h_sBsB', 'value': 4.0, 'vary': True, 'min': 0.0,
↪'fiducial': 4.0, 'prior_name': 'normal', 'mu': 4.0, 'sigma': 1.0, 'analytic': False}
theory.f_so = {'name': 'f_so', 'value': 0.03, 'vary': False, 'fiducial': 0.03, 'prior_
↪name': 'normal', 'mu': 0.04, 'sigma': 0.02, 'analytic': False}
theory.fcB = {'name': 'fcB', 'value': 0.08898809523809524, 'vary': False, 'min': 0,
↪'max': 1, 'expr': 'fs / (1 - fs) * (1 + fsB*(1./Nsat_mult - 1))', 'fiducial': 0.089,
↪ 'analytic': False}
theory.fs = {'name': 'fs', 'value': 0.104, 'vary': True, 'min': 0.0, 'max': 1.0,
↪'fiducial': 0.104, 'prior_name': 'uniform', 'lower': 0.0, 'upper': 0.25, 'analytic
↪': False}
theory.fsB = {'name': 'fsB', 'value': 0.4, 'vary': True, 'min': 0.0, 'max': 1,
↪'fiducial': 0.4, 'prior_name': 'uniform', 'lower': 0.0, 'upper': 1.0, 'analytic':␣
↪False}
theory.fsigma8 = {'name': 'fsigma8', 'value': 0.4758, 'vary': False, 'expr':
↪'f*sigma8_z', 'analytic': False}
theory.gamma_b1cB = {'name': 'gamma_b1cB', 'value': 0.4, 'vary': False, 'min': 0.0,
↪'max': 1.0, 'fiducial': 0.4, 'prior_name': 'normal', 'mu': 0.4, 'sigma': 0.2,
↪'analytic': False}
```

(continues on next page)

(continued from previous page)

```
theory.gamma_b1sA = {'name': 'gamma_b1sA', 'value': 1.45, 'vary': True, 'min': 1.0,
↪'fiducial': 1.45, 'prior_name': 'normal', 'mu': 1.45, 'sigma': 0.3, 'analytic':␣
↪False}
theory.gamma_b1sB = {'name': 'gamma_b1sB', 'value': 2.05, 'vary': True, 'min': 1.0,
↪'fiducial': 2.05, 'prior_name': 'normal', 'mu': 2.05, 'sigma': 0.3, 'analytic':␣
↪False}
theory.nbar = {'name': 'nbar', 'value': 0.0003117, 'vary': False, 'fiducial': 0.
↪0003117, 'analytic': False}
theory.sigma8_z = {'name': 'sigma8_z', 'value': 0.61, 'vary': True, 'fiducial': 0.61,
↪'prior_name': 'uniform', 'lower': 0.3, 'upper': 0.9, 'analytic': False}
theory.sigma_c = {'name': 'sigma_c', 'value': 1.0, 'vary': True, 'fiducial': 1.0,
↪'prior_name': 'uniform', 'lower': 0.0, 'upper': 3.0, 'analytic': False}
theory.sigma_sA = {'name': 'sigma_sA', 'value': 3.5, 'vary': True, 'fiducial': 3.5,
↪'prior_name': 'uniform', 'lower': 2.0, 'upper': 8.0, 'analytic': False}
theory.sigma_sB = {'name': 'sigma_sB', 'value': 5.0, 'vary': False, 'fiducial': 5.0,
↪'prior_name': 'uniform', 'lower': 3.0, 'upper': 10.0, 'analytic': False}
theory.sigma_so = {'name': 'sigma_so', 'value': 4.0, 'vary': False, 'fiducial': 4.0,
↪'prior_name': 'uniform', 'lower': 1.0, 'upper': 7, 'analytic': False}
#------------------------------------------------------------------------------

#------------------------------------------------------------------------------
# theory params
#------------------------------------------------------------------------------
theory.F_AP = {'name': 'F_AP', 'value': 1.0, 'vary': False, 'expr': 'alpha_par/alpha_
↪perp', 'analytic': False}
theory.N = {'name': 'N', 'value': 0.0, 'vary': False, 'fiducial': 0.0, 'prior_name':
↪'uniform', 'lower': -500.0, 'upper': 500.0, 'analytic': False}
theory.NcBs = {'name': 'NcBs', 'value': 40236.785434927464, 'vary': False, 'expr':
↪'f1h_cBs / (fcB*(1 - fs)*nbar)', 'fiducial': 45000.0, 'analytic': False}
theory.NsBsB = {'name': 'NsBsB', 'value': 128534.1756949072, 'vary': False, 'expr':
↪'f1h_sBsB / (fsB**2 * fs**2 * nbar) * (fcB*(1 - fs) - fs*(1-fsB))', 'fiducial':␣
↪94500.0, 'analytic': False}
theory.Nsat_mult = {'name': 'Nsat_mult', 'value': 2.4, 'vary': True, 'min': 2.0,
↪'fiducial': 2.4, 'prior_name': 'normal', 'mu': 2.4, 'sigma': 0.2, 'analytic': False}
theory.alpha = {'name': 'alpha', 'value': 1.0, 'vary': False, 'expr': '(alpha_perp**2␣
↪* alpha_par)**(1./3)', 'analytic': False}
theory.alpha_drag = {'name': 'alpha_drag', 'value': 1.0, 'vary': False, 'fiducial': 1.
↪0, 'analytic': False}
theory.alpha_par = {'name': 'alpha_par', 'value': 1.0, 'vary': True, 'fiducial': 1.0,
↪'prior_name': 'uniform', 'lower': 0.8, 'upper': 1.2, 'analytic': False}
theory.alpha_perp = {'name': 'alpha_perp', 'value': 1.0, 'vary': True, 'fiducial': 1.
↪0, 'prior_name': 'uniform', 'lower': 0.8, 'upper': 1.2, 'analytic': False}
theory.b1 = {'name': 'b1', 'value': 2.124276, 'vary': False, 'expr': '(1 - fs)*b1_c +␣
↪fs*b1_s', 'analytic': False}
theory.b1_c = {'name': 'b1_c', 'value': 1.9981383928571428, 'vary': False, 'expr':
↪'(1 - fcB)*b1_cA + fcB*b1_cB', 'analytic': False}
theory.b1_cA = {'name': 'b1_cA', 'value': 1.9, 'vary': True, 'fiducial': 1.9, 'prior_
↪name': 'uniform', 'lower': 1.2, 'upper': 2.5, 'analytic': False}
theory.b1_cB = {'name': 'b1_cB', 'value': 3.0028260869565218, 'vary': False, 'expr':
↪'(1-fsB)/(1+fsB*(1./Nsat_mult - 1)) * b1_sA +  (1 - (1-fsB)/(1+fsB*(1./Nsat_mult -␣
↪1))) * b1_sB', 'fiducial': 2.84, 'analytic': False}
theory.b1_s = {'name': 'b1_s', 'value': 3.2109999999999994, 'vary': False, 'expr':
↪'(1 - fsB)*b1_sA + fsB*b1_sB', 'analytic': False}
theory.b1_sA = {'name': 'b1_sA', 'value': 2.755, 'vary': False, 'expr': 'gamma_
↪b1sA*b1_cA', 'fiducial': 2.63, 'analytic': False}
theory.b1_sB = {'name': 'b1_sB', 'value': 3.8949999999999996, 'vary': False, 'expr':
↪'gamma_b1sB*b1_cA', 'fiducial': 3.62, 'analytic': False}
```

(continues on next page)

(continued from previous page)

```
theory.b1sigma8 = {'name': 'b1sigma8', 'value': 1.29580836, 'vary': False, 'expr':
↪'b1*sigma8_z', 'analytic': False}
theory.epsilon = {'name': 'epsilon', 'value': 0.0, 'vary': False, 'expr': '(alpha_
↪perp/alpha_par)**(-1./3) - 1.0', 'analytic': False}
theory.f = {'name': 'f', 'value': 0.78, 'vary': True, 'fiducial': 0.78, 'prior_name':
↪'uniform', 'lower': 0.6, 'upper': 1.0, 'analytic': False}
theory.f1h_cBs = {'name': 'f1h_cBs', 'value': 1.0, 'vary': False, 'min': 0, 'fiducial
↪': 1.0, 'prior_name': 'normal', 'mu': 1.0, 'sigma': 0.75, 'analytic': False}
theory.f1h_sBsB = {'name': 'f1h_sBsB', 'value': 4.0, 'vary': True, 'min': 0.0,
↪'fiducial': 4.0, 'prior_name': 'normal', 'mu': 4.0, 'sigma': 1.0, 'analytic': False}
theory.f_so = {'name': 'f_so', 'value': 0.03, 'vary': False, 'fiducial': 0.03, 'prior_
↪name': 'normal', 'mu': 0.04, 'sigma': 0.02, 'analytic': False}
theory.fcB = {'name': 'fcB', 'value': 0.08898809523809524, 'vary': False, 'min': 0,
↪'max': 1, 'expr': 'fs / (1 - fs) * (1 + fsB*(1./Nsat_mult - 1))', 'fiducial': 0.089,
↪ 'analytic': False}
theory.fs = {'name': 'fs', 'value': 0.104, 'vary': True, 'min': 0.0, 'max': 1.0,
↪'fiducial': 0.104, 'prior_name': 'uniform', 'lower': 0.0, 'upper': 0.25, 'analytic
↪': False}
theory.fsB = {'name': 'fsB', 'value': 0.4, 'vary': True, 'min': 0.0, 'max': 1,
↪'fiducial': 0.4, 'prior_name': 'uniform', 'lower': 0.0, 'upper': 1.0, 'analytic':␣
↪False}
theory.fsigma8 = {'name': 'fsigma8', 'value': 0.4758, 'vary': False, 'expr':
↪'f*sigma8_z', 'analytic': False}
theory.gamma_b1cB = {'name': 'gamma_b1cB', 'value': 0.4, 'vary': False, 'min': 0.0,
↪'max': 1.0, 'fiducial': 0.4, 'prior_name': 'normal', 'mu': 0.4, 'sigma': 0.2,
↪'analytic': False}
theory.gamma_b1sA = {'name': 'gamma_b1sA', 'value': 1.45, 'vary': True, 'min': 1.0,
↪'fiducial': 1.45, 'prior_name': 'normal', 'mu': 1.45, 'sigma': 0.3, 'analytic':␣
↪False}
theory.gamma_b1sB = {'name': 'gamma_b1sB', 'value': 2.05, 'vary': True, 'min': 1.0,
↪'fiducial': 2.05, 'prior_name': 'normal', 'mu': 2.05, 'sigma': 0.3, 'analytic':␣
↪False}
theory.nbar = {'name': 'nbar', 'value': 0.0003117, 'vary': False, 'fiducial': 0.
↪0003117, 'analytic': False}
theory.sigma8_z = {'name': 'sigma8_z', 'value': 0.61, 'vary': True, 'fiducial': 0.61,
↪'prior_name': 'uniform', 'lower': 0.3, 'upper': 0.9, 'analytic': False}
theory.sigma_c = {'name': 'sigma_c', 'value': 1.0, 'vary': True, 'fiducial': 1.0,
↪'prior_name': 'uniform', 'lower': 0.0, 'upper': 3.0, 'analytic': False}
theory.sigma_sA = {'name': 'sigma_sA', 'value': 3.5, 'vary': True, 'fiducial': 3.5,
↪'prior_name': 'uniform', 'lower': 2.0, 'upper': 8.0, 'analytic': False}
theory.sigma_sB = {'name': 'sigma_sB', 'value': 5.0, 'vary': False, 'fiducial': 5.0,
↪'prior_name': 'uniform', 'lower': 3.0, 'upper': 10.0, 'analytic': False}
theory.sigma_so = {'name': 'sigma_so', 'value': 4.0, 'vary': False, 'fiducial': 4.0,
↪'prior_name': 'uniform', 'lower': 1.0, 'upper': 7, 'analytic': False}
#------------------------------------------------------------------------------

#------------------------------------------------------------------------------
# theory params
#------------------------------------------------------------------------------
theory.F_AP = {'name': 'F_AP', 'value': 1.0, 'vary': False, 'expr': 'alpha_par/alpha_
↪perp', 'analytic': False}
theory.N = {'name': 'N', 'value': 0.0, 'vary': False, 'fiducial': 0.0, 'prior_name':
↪'uniform', 'lower': -500.0, 'upper': 500.0, 'analytic': False}
theory.NcBs = {'name': 'NcBs', 'value': 40236.785434927464, 'vary': False, 'expr':
↪'f1h_cBs / (fcB*(1 - fs)*nbar)', 'fiducial': 45000.0, 'analytic': False}
theory.NsBsB = {'name': 'NsBsB', 'value': 128534.1756949072, 'vary': False, 'expr':
↪'f1h_sBsB / (fsB**2 * fs**2 * nbar) * (fcB*(1 - fs) - fs*(1-fsB))', 'fiducial':␣
↪94500.0, 'analytic': False}
```

(continues on next page)

(continued from previous page)

```
theory.Nsat_mult = {'name': 'Nsat_mult', 'value': 2.4, 'vary': True, 'min': 2.0,
↪'fiducial': 2.4, 'prior_name': 'normal', 'mu': 2.4, 'sigma': 0.2, 'analytic': False}
theory.alpha = {'name': 'alpha', 'value': 1.0, 'vary': False, 'expr': '(alpha_perp**2␣
↪* alpha_par)**(1./3)', 'analytic': False}
theory.alpha_drag = {'name': 'alpha_drag', 'value': 1.0, 'vary': False, 'fiducial': 1.
↪0, 'analytic': False}
theory.alpha_par = {'name': 'alpha_par', 'value': 1.0, 'vary': True, 'fiducial': 1.0,
↪'prior_name': 'uniform', 'lower': 0.8, 'upper': 1.2, 'analytic': False}
theory.alpha_perp = {'name': 'alpha_perp', 'value': 1.0, 'vary': True, 'fiducial': 1.
↪0, 'prior_name': 'uniform', 'lower': 0.8, 'upper': 1.2, 'analytic': False}
theory.b1 = {'name': 'b1', 'value': 2.124276, 'vary': False, 'expr': '(1 - fs)*b1_c +␣
↪fs*b1_s', 'analytic': False}
theory.b1_c = {'name': 'b1_c', 'value': 1.9981383928571428, 'vary': False, 'expr':
↪'(1 - fcB)*b1_cA + fcB*b1_cB', 'analytic': False}
theory.b1_cA = {'name': 'b1_cA', 'value': 1.9, 'vary': True, 'fiducial': 1.9, 'prior_
↪name': 'uniform', 'lower': 1.2, 'upper': 2.5, 'analytic': False}
theory.b1_cB = {'name': 'b1_cB', 'value': 3.0028260869565218, 'vary': False, 'expr':
↪'(1-fsB)/(1+fsB*(1./Nsat_mult - 1)) * b1_sA +  (1 - (1-fsB)/(1+fsB*(1./Nsat_mult -␣
↪1))) * b1_sB', 'fiducial': 2.84, 'analytic': False}
theory.b1_s = {'name': 'b1_s', 'value': 3.2109999999999994, 'vary': False, 'expr':
↪'(1 - fsB)*b1_sA + fsB*b1_sB', 'analytic': False}
theory.b1_sA = {'name': 'b1_sA', 'value': 2.755, 'vary': False, 'expr': 'gamma_
↪b1sA*b1_cA', 'fiducial': 2.63, 'analytic': False}
theory.b1_sB = {'name': 'b1_sB', 'value': 3.8949999999999996, 'vary': False, 'expr':
↪'gamma_b1sB*b1_cA', 'fiducial': 3.62, 'analytic': False}
theory.b1sigma8 = {'name': 'b1sigma8', 'value': 1.29580836, 'vary': False, 'expr':
↪'b1*sigma8_z', 'analytic': False}
theory.epsilon = {'name': 'epsilon', 'value': 0.0, 'vary': False, 'expr': '(alpha_
↪perp/alpha_par)**(-1./3) - 1.0', 'analytic': False}
theory.f = {'name': 'f', 'value': 0.78, 'vary': True, 'fiducial': 0.78, 'prior_name':
↪'uniform', 'lower': 0.6, 'upper': 1.0, 'analytic': False}
theory.f1h_cBs = {'name': 'f1h_cBs', 'value': 1.0, 'vary': False, 'min': 0, 'fiducial
↪': 1.0, 'prior_name': 'normal', 'mu': 1.0, 'sigma': 0.75, 'analytic': False}
theory.f1h_sBsB = {'name': 'f1h_sBsB', 'value': 4.0, 'vary': True, 'min': 0.0,
↪'fiducial': 4.0, 'prior_name': 'normal', 'mu': 4.0, 'sigma': 1.0, 'analytic': False}
theory.f_so = {'name': 'f_so', 'value': 0.03, 'vary': False, 'fiducial': 0.03, 'prior_
↪name': 'normal', 'mu': 0.04, 'sigma': 0.02, 'analytic': False}
theory.fcB = {'name': 'fcB', 'value': 0.08898809523809524, 'vary': False, 'min': 0,
↪'max': 1, 'expr': 'fs / (1 - fs) * (1 + fsB*(1./Nsat_mult - 1))', 'fiducial': 0.089,
↪ 'analytic': False}
theory.fs = {'name': 'fs', 'value': 0.104, 'vary': True, 'min': 0.0, 'max': 1.0,
↪'fiducial': 0.104, 'prior_name': 'uniform', 'lower': 0.0, 'upper': 0.25, 'analytic
↪': False}
theory.fsB = {'name': 'fsB', 'value': 0.4, 'vary': True, 'min': 0.0, 'max': 1,
↪'fiducial': 0.4, 'prior_name': 'uniform', 'lower': 0.0, 'upper': 1.0, 'analytic':␣
↪False}
theory.fsigma8 = {'name': 'fsigma8', 'value': 0.4758, 'vary': False, 'expr':
↪'f*sigma8_z', 'analytic': False}
theory.gamma_b1cB = {'name': 'gamma_b1cB', 'value': 0.4, 'vary': False, 'min': 0.0,
↪'max': 1.0, 'fiducial': 0.4, 'prior_name': 'normal', 'mu': 0.4, 'sigma': 0.2,
↪'analytic': False}
theory.gamma_b1sA = {'name': 'gamma_b1sA', 'value': 1.45, 'vary': True, 'min': 1.0,
↪'fiducial': 1.45, 'prior_name': 'normal', 'mu': 1.45, 'sigma': 0.3, 'analytic':␣
↪False}
theory.gamma_b1sB = {'name': 'gamma_b1sB', 'value': 2.05, 'vary': True, 'min': 1.0,
↪'fiducial': 2.05, 'prior_name': 'normal', 'mu': 2.05, 'sigma': 0.3, 'analytic':␣
↪False}
```

(continues on next page)

(continued from previous page)

```
theory.nbar = {'name': 'nbar', 'value': 0.0003117, 'vary': False, 'fiducial': 0.
↪0003117, 'analytic': False}
theory.sigma8_z = {'name': 'sigma8_z', 'value': 0.61, 'vary': True, 'fiducial': 0.61,
↪'prior_name': 'uniform', 'lower': 0.3, 'upper': 0.9, 'analytic': False}
theory.sigma_c = {'name': 'sigma_c', 'value': 1.0, 'vary': True, 'fiducial': 1.0,
↪'prior_name': 'uniform', 'lower': 0.0, 'upper': 3.0, 'analytic': False}
theory.sigma_sA = {'name': 'sigma_sA', 'value': 3.5, 'vary': True, 'fiducial': 3.5,
↪'prior_name': 'uniform', 'lower': 2.0, 'upper': 8.0, 'analytic': False}
theory.sigma_sB = {'name': 'sigma_sB', 'value': 5.0, 'vary': False, 'fiducial': 5.0,
↪'prior_name': 'uniform', 'lower': 3.0, 'upper': 10.0, 'analytic': False}
theory.sigma_so = {'name': 'sigma_so', 'value': 4.0, 'vary': False, 'fiducial': 4.0,
↪'prior_name': 'uniform', 'lower': 1.0, 'upper': 7, 'analytic': False}
#------------------------------------------------------------------------------

#------------------------------------------------------------------------------
# theory params
#------------------------------------------------------------------------------
theory.F_AP = {'name': 'F_AP', 'value': 1.0, 'vary': False, 'expr': 'alpha_par/alpha_
↪perp', 'analytic': False}
theory.N = {'name': 'N', 'value': 0.0, 'vary': False, 'fiducial': 0.0, 'prior_name':
↪'uniform', 'lower': -500.0, 'upper': 500.0, 'analytic': False}
theory.NcBs = {'name': 'NcBs', 'value': 40236.785434927464, 'vary': False, 'expr':
↪'f1h_cBs / (fcB*(1 - fs)*nbar)', 'fiducial': 45000.0, 'analytic': False}
theory.NsBsB = {'name': 'NsBsB', 'value': 128534.1756949072, 'vary': False, 'expr':
↪'f1h_sBsB / (fsB**2 * fs**2 * nbar) * (fcB*(1 - fs) - fs*(1-fsB))', 'fiducial':␣
↪94500.0, 'analytic': False}
theory.Nsat_mult = {'name': 'Nsat_mult', 'value': 2.4, 'vary': True, 'min': 2.0,
↪'fiducial': 2.4, 'prior_name': 'normal', 'mu': 2.4, 'sigma': 0.2, 'analytic': False}
theory.alpha = {'name': 'alpha', 'value': 1.0, 'vary': False, 'expr': '(alpha_perp**2␣
↪* alpha_par)**(1./3)', 'analytic': False}
theory.alpha_drag = {'name': 'alpha_drag', 'value': 1.0, 'vary': False, 'fiducial': 1.
↪0, 'analytic': False}
theory.alpha_par = {'name': 'alpha_par', 'value': 1.0, 'vary': True, 'fiducial': 1.0,
↪'prior_name': 'uniform', 'lower': 0.8, 'upper': 1.2, 'analytic': False}
theory.alpha_perp = {'name': 'alpha_perp', 'value': 1.0, 'vary': True, 'fiducial': 1.
↪0, 'prior_name': 'uniform', 'lower': 0.8, 'upper': 1.2, 'analytic': False}
theory.b1 = {'name': 'b1', 'value': 2.124276, 'vary': False, 'expr': '(1 - fs)*b1_c +␣
↪fs*b1_s', 'analytic': False}
theory.b1_c = {'name': 'b1_c', 'value': 1.9981383928571428, 'vary': False, 'expr':
↪'(1 - fcB)*b1_cA + fcB*b1_cB', 'analytic': False}
theory.b1_cA = {'name': 'b1_cA', 'value': 1.9, 'vary': True, 'fiducial': 1.9, 'prior_
↪name': 'uniform', 'lower': 1.2, 'upper': 2.5, 'analytic': False}
theory.b1_cB = {'name': 'b1_cB', 'value': 3.0028260869565218, 'vary': False, 'expr':
↪'(1-fsB)/(1+fsB*(1./Nsat_mult - 1)) * b1_sA +  (1 - (1-fsB)/(1+fsB*(1./Nsat_mult -␣
↪1))) * b1_sB', 'fiducial': 2.84, 'analytic': False}
theory.b1_s = {'name': 'b1_s', 'value': 3.2109999999999994, 'vary': False, 'expr':
↪'(1 - fsB)*b1_sA + fsB*b1_sB', 'analytic': False}
theory.b1_sA = {'name': 'b1_sA', 'value': 2.755, 'vary': False, 'expr': 'gamma_
↪b1sA*b1_cA', 'fiducial': 2.63, 'analytic': False}
theory.b1_sB = {'name': 'b1_sB', 'value': 3.8949999999999996, 'vary': False, 'expr':
↪'gamma_b1sB*b1_cA', 'fiducial': 3.62, 'analytic': False}
theory.b1sigma8 = {'name': 'b1sigma8', 'value': 1.29580836, 'vary': False, 'expr':
↪'b1*sigma8_z', 'analytic': False}
theory.epsilon = {'name': 'epsilon', 'value': 0.0, 'vary': False, 'expr': '(alpha_
↪perp/alpha_par)**(-1./3) - 1.0', 'analytic': False}
theory.f = {'name': 'f', 'value': 0.78, 'vary': True, 'fiducial': 0.78, 'prior_name':
↪'uniform', 'lower': 0.6, 'upper': 1.0, 'analytic': False}
```

(continues on next page)

(continued from previous page)

```
theory.f1h_cBs = {'name': 'f1h_cBs', 'value': 1.0, 'vary': False, 'min': 0, 'fiducial
↪': 1.0, 'prior_name': 'normal', 'mu': 1.0, 'sigma': 0.75, 'analytic': False}
theory.f1h_sBsB = {'name': 'f1h_sBsB', 'value': 4.0, 'vary': True, 'min': 0.0,
↪'fiducial': 4.0, 'prior_name': 'normal', 'mu': 4.0, 'sigma': 1.0, 'analytic': False}
theory.f_so = {'name': 'f_so', 'value': 0.03, 'vary': False, 'fiducial': 0.03, 'prior_
↪name': 'normal', 'mu': 0.04, 'sigma': 0.02, 'analytic': False}
theory.fcB = {'name': 'fcB', 'value': 0.08898809523809524, 'vary': False, 'min': 0,
↪'max': 1, 'expr': 'fs / (1 - fs) * (1 + fsB*(1./Nsat_mult - 1))', 'fiducial': 0.089,
↪ 'analytic': False}
theory.fs = {'name': 'fs', 'value': 0.104, 'vary': True, 'min': 0.0, 'max': 1.0,
↪'fiducial': 0.104, 'prior_name': 'uniform', 'lower': 0.0, 'upper': 0.25, 'analytic
↪': False}
theory.fsB = {'name': 'fsB', 'value': 0.4, 'vary': True, 'min': 0.0, 'max': 1,
↪'fiducial': 0.4, 'prior_name': 'uniform', 'lower': 0.0, 'upper': 1.0, 'analytic':␣
↪False}
theory.fsigma8 = {'name': 'fsigma8', 'value': 0.4758, 'vary': False, 'expr':
↪'f*sigma8_z', 'analytic': False}
theory.gamma_b1cB = {'name': 'gamma_b1cB', 'value': 0.4, 'vary': False, 'min': 0.0,
↪'max': 1.0, 'fiducial': 0.4, 'prior_name': 'normal', 'mu': 0.4, 'sigma': 0.2,
↪'analytic': False}
theory.gamma_b1sA = {'name': 'gamma_b1sA', 'value': 1.45, 'vary': True, 'min': 1.0,
↪'fiducial': 1.45, 'prior_name': 'normal', 'mu': 1.45, 'sigma': 0.3, 'analytic':␣
↪False}
theory.gamma_b1sB = {'name': 'gamma_b1sB', 'value': 2.05, 'vary': True, 'min': 1.0,
↪'fiducial': 2.05, 'prior_name': 'normal', 'mu': 2.05, 'sigma': 0.3, 'analytic':␣
↪False}
theory.nbar = {'name': 'nbar', 'value': 0.0003117, 'vary': False, 'fiducial': 0.
↪0003117, 'analytic': False}
theory.sigma8_z = {'name': 'sigma8_z', 'value': 0.61, 'vary': True, 'fiducial': 0.61,
↪'prior_name': 'uniform', 'lower': 0.3, 'upper': 0.9, 'analytic': False}
theory.sigma_c = {'name': 'sigma_c', 'value': 1.0, 'vary': True, 'fiducial': 1.0,
↪'prior_name': 'uniform', 'lower': 0.0, 'upper': 3.0, 'analytic': False}
theory.sigma_sA = {'name': 'sigma_sA', 'value': 3.5, 'vary': True, 'fiducial': 3.5,
↪'prior_name': 'uniform', 'lower': 2.0, 'upper': 8.0, 'analytic': False}
theory.sigma_sB = {'name': 'sigma_sB', 'value': 5.0, 'vary': False, 'fiducial': 5.0,
↪'prior_name': 'uniform', 'lower': 3.0, 'upper': 10.0, 'analytic': False}
theory.sigma_so = {'name': 'sigma_so', 'value': 4.0, 'vary': False, 'fiducial': 4.0,
↪'prior_name': 'uniform', 'lower': 1.0, 'upper': 7, 'analytic': False}
#------------------------------------------------------------------------------
```

1.11.3 Configuring the Theory

Even if the user chooses to use the default theory parameters returned by *GalaxySpectrum.default_params()*, there are several areas where the user should customize the parameters for the specific data set being fit. We will described these customizations in this section.

Changing the Parametrization

The recommended way to make changes to the model parametrization is by adjusting the parameters in the default `pyRSD.rsdfit.theory.parameters.ParameterSet` object. The user should get the default parameters from the `default_params()` function and then make the desired changes, before writing the parameters to a file.

For example, to fix the satellite fraction and only use 12 free parameters in the fitting procedure, one can simply do

```
In [1]: from pyRSD.rsd import GalaxySpectrum

In [2]: model = GalaxySpectrum()

# default params
In [3]: params = model.default_params()

# fix the satellite fraction
In [4]: params['fs'].vary = False

# write new configuration to a file
In [5]: params.to_file('params.dat', mode='a')
```

The Sample Number Density

The model requires the number density of the sample to properly account for 1-halo terms. This number density is assumed to be constant and in the case of a number density that varies with redshift, the value can be thought of an average number density.

Note: Typically, the number density value used is the inverse of the sample shot noise.

The value of the `nbar` parameter should be updated by the user, as

```
# get the default params
In [6]: params = model.default_params()

# this is the default nbar value
In [7]: print(params['nbar'])
<Parameter 'nbar'                value=0.0003117       (fixed, fiducial)   no prior>

# change to the right value
In [8]: params['nbar'].value = 4e-5 # in units of (Mpc/h)^{-3}
```

Configuring the GalaxySpectrum Model

The *GalaxySpectrum* model has several configuration parameters, which can be passed to `__init__()` function. Most of these parameters have sensible default values that should not be changed by the user. However, there are a few that should be changed based on the data set being fit. The most important are the sample redshift and the cosmological parameters.

The redshift of the sample is taken to be constant. For samples that have a number density varying with redshift, the redshift should be set to the effective redshift of the sample. In the parameter file passed to `rsdfit` the redshift can be set as:

```
model.z = 0.55
```

The cosmological parameters can also be specified by the user in the parameter file – they set the shape of the linear power spectrum which is the input to the *GalaxySpectrum* model. However, if an already initialized model is being passed to the `rsdfit` command via the `-m` flag, then the user doesn't necessarily need to specify the cosmology again.

The cosmological parameters can be specified in a number of ways:

1. **the name of a file**

The cosmological parameters can be read from a parameter file. See the `pyRSD/data/params` directory for example parameter files.

2. **the name of a builtin cosmology**

   The name of a builtin cosmology, such as `Planck15` or `WMAP9`

3. **a dictionary of parameters**

   A dictionary of key/value pairs, such as `Ob0`, `Om0`, etc, which will be passed to the `pyRSD.rsd.cosmology.Cosmology.__init__` function.

The cosmology parameters can be specified in the parameter file by setting the `model.params` parameter. For example, to use the WMAP9 parameter set, simply specify the following in the parameter file:

```
model.params = "WMAP9"
```

Additionally, there are other model configuraton parameters that can be set in the parameter file, as long as they are prefixed with `model.`. The `config` attribute of the *GalaxySpectrum* object gives these values

```
In [9]: from pyRSD.rsd import GalaxySpectrum

In [10]: model = GalaxySpectrum()

In [11]: for k in sorted(model.config):
   ....:     print("%s = %s" %(k, str(model.config[k])))
   ....:
Nk = 200
Pdv_model_type = jennings
correct_mu2 = False
correct_mu4 = False
fog_model = modified_lorentzian
include_2loop = False
interpolate = True
k0_low = 0.005
kmax = 0.5
kmin = 0.001
linear_power_file = None
max_mu = 4
params = {'H0': 67.74, 'Om0': 0.3075, 'sigma8': 0.8159, 'n_s': 0.9667, 'flat': True,
↪'w0': -1.0, 'name': 'Planck15', 'Ob0': 0.0486, 'Tcmb0': 2.7255, 'Neff': 3.046, 'm_nu
↪': array([0.  , 0.  , 0.06])}
transfer_fit = CLASS
use_P00_model = True
use_P01_model = True
use_P11_model = True
use_Pdv_model = True
use_Phm_model = True
use_Pvv_model = True
use_mean_bias = False
use_so_correction = False
use_tidal_bias = False
use_vlah_biasing = True
vel_disp_from_sims = False
z = 0.0
```

1.11.4 API

Galaxy Power Theory

class pyRSD.rsdfit.theory.**GalaxyPowerTheory**(*param_file*, *model=None*, *extra_param_file=None*, *kmin=None*, *kmax=None*)

A class representing a theory for computing a redshift-space galaxy power spectrum.

It handles the dependencies between model parameters and the evaluation of the model itself.

check(*return_errors=False*)

Check the values of all parameters. Here, *check* means that each parameter is within its bounds and the prior is not infinity

If *return_errors* = *True*, return the error messages as well

dlnprior

Return the derivative of the log prior for all "free" parameters

evaluate_grad_lnlike(*theta*, *data*, *pool=None*, *epsilon=0.0001*, *numerical=False*, *theory_decorator={}*)

Evaluate the gradient.

free_fiducial

Return an array of the fiducial free parameters

get_kmu_pairs(*data*)

Compute the flattened `k` and `mu` values needed to evaluate the theory prediction, given the transfer functions defined by `data`.

This also computes the slices needed to recover the pairs for individual transfer functions.

inverse_scale(*theta*)

Inverse scale the free parameters, using the priors to define the scaling transformation

lnprior

Return the log prior for all "free" parameters as the sum of the priors of each individual parameter

lnprior_constrained

Return the log prior for constrained parameters as the sum of the priors of each individual parameter

lnprior_free

Return the log prior for free parameters as the sum of the priors of each individual parameter

model_callable(*data*, *theory_decorator={}*)

Return the flattened theory prediction corresponding to the statistics in the `data` object.

Parameters **data** : PowerData

the data class, which tells the theory which statistics to compute and what basis to evaluate the theory in, e.g., multipoles, wedges, window-convoled, etc

theory_decorator : dict, optional

dictionary of decorators to apply to the theory predictions for individual data statistics

ndim

Returns the number of free parameters, i.e., the *dimension* of the theory

pkmu_gradient

Return the P(k,mu) gradient class

preserve(*theta*)
Context manager that preserves the state of the model upon exiting the context by first saving and then restoring it

scale(*theta*)
Scale the (unscaled) free parameters, using the priors to define the scaling transformation

scale_gradient(*grad*)
Scale the gradient with respect to the unscaled free parameters, using the priors to define the scaling transformation

This returns df / dxprime where xprime is the scaled param vector

set_free_parameters(*theta*)
Given an array of values *theta*, set the free parameters of attr:*fit_params*

Returns **valid_model** : bool

return *True/False* flag indicating if we were able to successfully set the free parameters and update the model

Notes

If any free parameter values are outside their bounds, the model will not be updated and *False* will be returned

to_file(*filename*, *mode='w'*)
Save the parameters of this theory in a file

Parameter

class pyRSD.rsdfit.parameters.**Parameter**(*name=None*, *value=None*, *vary=False*, *min=None*, *max=None*, *expr=None*, ***kwargs*)
A subclass of `lmfit.parameter.Parameter` to represent a generic parameter.

The added functionality is largely the ability to associate a prior with the parameter. Currently, the prior can either be a uniform or normal distribution.

Parameters **name** : str

The string name of the parameter. Must be supplied

description : str, optional

The string giving the parameter description

value : object, optional

The value. This can be a float, in which case the other attributes have meaning; otherwise the class just stores the object value. Default is *Parameter.fiducial*

fiducial : object, optional

A fiducial value to store for this parameter. Default is *None*.

vary : bool, optional

Whether to vary this parameter. Default is *False*.

min : float, optional

The minimum allowed limit for this parameter, for use in excluding "disallowed" parameter spaces. Default is *None*

min : float, optional

The maximum allowed limit for this parameter, for use in excluding "disallowed" parameter spaces. Default is *None*

expr : {str, callable}, optional

If a *str*, this gives a mathematical expression used to constrain the value during the fit. Otherwise, the function will be called to evaluate the parameter's value. Default is *None*

prior_name : {'uniform', 'normal'}, optional

Use either a uniform or normal prior for this parameter. Default is no prior.

lower, upper : floats, optional

The bounds of the uniform prior to use

mu, sigma : floats, optional

The mean and std deviation of the normal prior to use

analytic
If *True*, use an analytic approximation for *Uniform* priors and the min/max bounds

bounded
Parameter is bounded if either *min* or *max* is defined

constrained
Parameter is constrained if the *Parameter.expr == None*

description
The parameter description

dlnprior
Return the derivative of the log of the prior, which is either a uniform or normal prior. If the current value is outside *Parameter.min* or *Parameter.max*, return *numpy.inf*

dtype
The data type of the parameter

expr
Return the mathematical expression used to constrain the value during the fit.

fiducial
A fiducial value associated with this parameter

get_value_from_prior(*size=1*)
Get random values from the prior, of size *size*

has_fiducial
Whether the parameter has a fiducial value defined

has_prior
Whether the parameter has a prior defined

lnprior
Return log of the prior, which is either a uniform or normal prior. If the current value is outside *Parameter.min* or *Parameter.max*, return *numpy.inf*

loc
The *loc* of the parameter as determined from the prior

lower
The lower limit of the uniform prior

max
The maximum allowed value (exclusive)

max_bound
The distribution representing the minimum bound

min
The minimum allowed value (inclusive)

min_bound
The distribution representing the minimum bound

mu
The mean value of the normal prior

output_value
Explicit value for output – defaults to *value*

prior
The prior distribution

prior_name
The name of the prior distribution

scale
The *scale* of the parameter as determined from the prior

scale_gradient(*val*)
Return scaling factor for gradient.

Parameters **val: float**

Numerical Parameter value.

Returns float

Scaling factor for gradient the according to Minuit-style transformation.

set(*value=None*, *vary=None*, *min=None*, *max=None*, *expr=None*, *brute_step=None*)
Set or update Parameter attributes.

Parameters **value** : float, optional

Numerical Parameter value.

vary : bool, optional

Whether the Parameter is varied during a fit.

min : float, optional

Lower bound for value. To remove a lower bound you must use *-numpy.inf*.

max : float, optional

Upper bound for value. To remove an upper bound you must use *numpy.inf*.

expr : str, optional

Mathematical expression used to constrain the value during the fit. To remove a constraint you must supply an empty string.

brute_step : float, optional

Step size for grid points in the *brute* method. To remove the step size you must use `0`.

Notes

Each argument to *set()* has a default value of *None*, which will leave the current value for the attribute unchanged. Thus, to lift a lower or upper bound, passing in *None* will not work. Instead, you must set these to *-numpy.inf* or *numpy.inf*, as with:

```
par.set(min=None)         # leaves lower bound unchanged
par.set(min=-numpy.inf)   # removes lower bound
```

Similarly, to clear an expression, pass a blank string, (not `None`!) as with:

```
par.set(expr=None)   # leaves expression unchanged
par.set(expr='')     # removes expression
```

Explicitly setting a value or setting *vary=True* will also clear the expression.

Finally, to clear the brute_step size, pass `0`, not `None`:

```
par.set(brute_step=None)   # leaves brute_step unchanged
par.set(brute_step=0)      # removes brute_step
```

set_expr_eval (*evaluator*)
Set expression evaluator instance.

setup_bounds ()
Set up Minuit-style internal/external parameter transformation of min/max bounds.

As a side-effect, this also defines the self.from_internal method used to re-calculate self.value from the internal value, applying the inverse Minuit-style transformation. This method should be called prior to passing a Parameter to the user-defined objective function.

This code borrows heavily from JJ Helmus' leastsqbound.py

Returns **_val** : float

The internal value for parameter from self.value (which holds the external, user-expected value). This internal value should actually be used in a fit.

sigma
The standard deviation of the normal prior

to_dict (*output=False*)
Convert the *Parameter* object to a dictionary

update (***kwargs*)
Update the attributes of the Parameter

upper
The upper limit of the uniform prior

value
Return the numerical value of the Parameter, with bounds applied.

within_bounds (*x=None*)
Returns *True* if the specified value is within the (min, max) bounds and if the prior is uniform, within the lower/upper values of the prior

ParameterSet

class pyRSD.rsdfit.parameters.**ParameterSet**(**args*, ***kwargs*)

A subclass of `lmfit.parameter.Parameters` that adds the ability to update values based on constraints in place

add(*name*, ***kwargs*)

Add a *Parameter* with *name* to the ParameterSet with the specified *value*.

add_many(**parlist*)

Add many parameters, using a sequence of tuples.

Parameters **parlist** : sequence of tuple or *Parameter*

A sequence of tuples, or a sequence of *Parameter* instances. If it is a sequence of tuples, then each tuple must contain at least the name. The order in each tuple must be *(name, value, vary, min, max, expr, brute_step)*.

Examples

```
>>>  params = Parameters()
# add with tuples: (NAME VALUE VARY MIN  MAX  EXPR  BRUTE_STEP)
>>> params.add_many(('amp',   10, True, None, None, None, None),
...                 ('cen',   4, True,  0.0, None, None, None),
...                 ('wid',   1, False, None, None, None, None),
...                 ('frac', 0.5))
# add a sequence of Parameters
>>> f = Parameter('par_f', 100)
>>> g = Parameter('par_g',  2.)
>>> params.add_many(f, g)
```

clear() → None. Remove all items from od.

constrained

Return a list of the constrained *Parameter* objects.

constrained_dtype

The list of 'constrained' data types for the structured array

constrained_names

Return the constrained parameter names. *Constrained* means that *Parameter.constrained = True*

constrained_values

Return the constrained parameter values.

constraint_derivative(*name*, *wrt*, *theta=None*)

Return the constraint derivative at the specified values of free parameters. If *theta* is not specified, the current values are used.

This returns d'name' / d'wrt'

Parameters **name** : str

the name of the constrained derivative to take the derivative of

wrt : str

the name of the free parameter to the derivative with respect to

theta : array_like, optional

the array of free parameters to evaluate the derivative at

Returns **grad** : float

the value of the gradient

copy()
Parameters.copy() should always be a deepcopy.

delayed_asteval()
Context manager to handle delaying asteval evaluation

dump(*fp*, ***kws*)
Write JSON representation of Parameters to a file-like object.

Parameters **fp** : file-like object

An open and `.write()`-supporting file-like object.

****kws** : optional

Keyword arguments that are passed to *dumps()*.

Returns None or int

Return value from *fp.write()*. None for Python 2.7 and the number of characters written in Python 3.

See also:

dump, *load*, `json.dump`

dumps(***kws*)
Represent Parameters as a JSON string.

Parameters ****kws** : optional

Keyword arguments that are passed to *json.dumps()*.

Returns str

JSON string representation of Parameters.

See also:

dump, *loads*, *load*, `json.dumps`

free
Return a list of the free *Parameter* objects. *Free* means that *Parameter.vary* = *True* and *Parameter.constrained* = *False*

free_dtype
The list of 'free' data types for the structured array

free_names
Return the free parameter names. *Free* means that *Parameter.vary* = *True* and *Parameter.constrained* = *False*

free_values
Return the free parameter values. *Free* means that *Parameter.vary* = *True* and *Parameter.constrained* = *False*

classmethod from_file(*filename*, *tags=[]*)
Read a file and return a *collections.defaultdict* with the keys given by *tags* and the values given by a *lmfit.Parameters* object

Parameters **filename** : str

the name of the file to read parameters from

tags : list, optional

list of any parameter tags to specifically seach for

fromkeys (*S*[, *v*]) → New ordered dictionary with keys from S.
If not specified, the value defaults to None.

get (*name*, *default=None*)
Mirrors the *dict.get()* method behavior, but returns the parameter values

items () → a set-like object providing a view on D's items

keys () → a set-like object providing a view on D's keys

load (*fp*, ***kws*)
Load JSON representation of Parameters from a file-like object.

Parameters **fp** : file-like object

An open and `.read()`-supporting file-like object.

****kws** : optional

Keyword arguments that are passed to *loads()*.

Returns `Parameters`

Updated Parameters loaded from *fp*.

See also:

dump, *loads*, `json.load`

loads (*s*, ***kws*)
Load Parameters from a JSON string.

Parameters ****kws** : optional

Keyword arguments that are passed to *json.loads()*.

Returns `Parameters`

Updated Parameters from the JSON string.

See also:

dump, *dumps*, *load*, `json.loads`

Notes

Current Parameters will be cleared before loading the data from the JSON string.

locs
The locs of the free parameters

move_to_end ()
Move an existing element to the end (or beginning if last==False).

Raises KeyError if the element does not exist. When last=True, acts like a fast version of self[key]=self.pop(key).

pop (*k*[, *d*]) → v, remove specified key and return the corresponding
value. If key is not found, d is returned if given, otherwise KeyError is raised.

popitem(*$self*, */*, *last=True*)
Remove and return a (key, value) pair from the dictionary.

Pairs are returned in LIFO order if last is true or FIFO order if false.

prepare_params()
Prepare the parameters by parsing the dependencies. We initialize the `ast` classes in order to evaluate any constrained parameters

pretty_print(*oneline=False, colwidth=8, precision=4, fmt='g', columns=['value', 'min', 'max', 'stderr', 'vary', 'expr', 'brute_step']*)
Pretty-print of parameters data.

Parameters **oneline** : bool, optional

If True prints a one-line parameters representation (default is False).

colwidth : int, optional

Column width for all columns specified in `columns`.

precision : int, optional

Number of digits to be printed after floating point.

fmt : {'g', 'e', 'f'}, optional

Single-character numeric formatter. Valid values are: 'f' floating point, 'g' floating point and exponential, or 'e' exponential.

columns : `list` of `str`, optional

List of *`Parameter`* attribute names to print.

pretty_repr(*oneline=False*)
Return a pretty representation of a Parameters class.

Parameters **oneline** : bool, optional

If True prints a one-line parameters representation (default is False).

Returns s: str

Parameters representation.

register_function(*name*, *function*)
Register a function in the `symtable` of the `asteval` attribute

scales
The scales of the free parameters

setdefault($k[, d]$) → od.get(k,d), also set od[k]=d if k not in od

to_dict()
Convert the parameter set to a dictionary using (*name*, *value*) as the (key, value) pairs

to_file(*filename*, *prefix=None*, *mode='w'*, *header_name=None*, *footer=False*, *as_dict=True*)
Output the *ParameterSet* to a file, using the mode specified. Optionally, add a header and/or footer to make it look nice.

If *as_dict* = *True*, output the parameter as a dictionary, otherwise just output the value

Parameters **filename** : str

the name of the file to write

prefix : str, optional

include a prefix when writing out parameter keys

mode : str, optional

the file mode, i.e., 'w' to write, 'a' to append

header_name : str, optional

write out a comment header first

footer : bool, optional

whether to write out a footer comment

as_dict : bool, optional

write out Parameter objects as a dictionary, instead of just the value

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

update_constraints()
Update all constrained parameters, checking that dependencies are evaluated as needed.

update_fiducial()
Update the fiducial values of the constrained parameters

update_param(**name*, ***kwargs*)
Update the *Parameter* specified by *name* with the keyword arguments provided

update_values(***kwargs*)
Update the values of all parameters, checking that dependencies are evaluated as needed.

Parameters ***args** : tuple of str

the parameter names for which we want to update the parameter constraints

****kwargs :**

update the values of these parameters, and then update the constraints then depend on these parameters, only if the value of the parameters changed

values() → an object providing a view on D's values

valuesdict()
Return an ordered dictionary of parameter values.

Returns OrderedDict

An ordered dictionary of `name`:`value` pairs for each Parameter.

1.12 Running the Fits

Once the data, covariance matrix, and theory parameters have be set, the user can run a likelihood analysis to find the best-fitting theory parameters. This can be done using the MCMC technique or nonlinear optimization using the LBFGS algorithm. In this section, we describe how to run both algorithms with the `rsdfit` executable, as well as the configuration options for both algorithms.

1.12.1 Overview

The main class that is responsible for handling the data statistics, as well as the covariance matrix used during parameter estimation, is the *pyRSD.rsdfit.FittingDriver* class. This class combines data and theory to run a Bayesian likelihood analysis.

Information about the parameters that are needed to initialize the *FittingDriver* class can be found by using the `FittingDriver.help()` function,

```
In [1]: from pyRSD.rsdfit import FittingDriver

# print out the help message for the parameters needed
In [2]: FittingDriver.help()
Initialization Parameters for FittingDriver
-------------------------------------------------
burnin :

        An integer specifying the number of MCMC steps to consider as part
        of the "burn-in" period when saving the results

        This doesn't affect the parameter estimation at all; it simply
        changes what best-fit parameters are printed to the screen when
        the fitting is over. All iterations are saved to file regardless of
        the value of this parameter.


        Default: 0

epsilon :

        The Gelman-Rubin convergence criterion


        Default: 0.02

init_from :

        How to initialize the optimization; can be 'nlopt', 'fiducial',
        'prior', or 'result'; default is None


        Default: fiducial

init_scatter :

        The percentage of additional scatter to add to the initial fitting
        parameters; default is 0


        Default: 0

lbfgs_epsilon :

        The step-size for derivatives in LBFGS; default is 1e-4

        A dictionary can supplied where keys specify the value to use
        for specific free parameters
```

(continues on next page)

(continued from previous page)

```
        Default: 0.0001

lbfgs_numerical :

        If `True`, evaluate gradients of P(k,mu) numerically using finite difference


        Default: False

lbfgs_numerical_from_lnlike :

        If `True`, evaluate the gradient by taking the numerical derivative
        of :func:`minus_lnlike`


        Default: False

lbfgs_options :

        Configuration options to pass to the LBFGS solver


        Default: {'factr': 100000.0, 'gtol': 1e-05}

lbfgs_use_priors :

        Whether to use priors when solving with the LBFGS algorithm; default
        is ``True``


        Default: True

start_from :

        The name of a result file to initialize the optimization from


        Default: None

test_convergence :

        Whether to test for convergence of the parameter fits

        For MCMC fits, the Gelman-Rubin criteria is used as specified
        by the 'epsilon' parameter. For the LBFGS solver, the
        convergence criteria is set by the 'lbfgs_options' attribute.


        Default: False

theory_decorator :

        A decorator to wrap the default theory output.

        Users can use this to compute linear combinations of multipoles, for
        example.
```

(continues on next page)

(continued from previous page)

```
        Default: {}

tracer_type :

        The type of tracer, either 'galaxy' or 'quasar'

        Default: galaxy
```

These parameters should be specified in the parameter file that is passed to the `rsdfit` executable and the names of the parameters should be prefixed with the `driver.` prefix. In our example parameter file discussed previously, we the following parameters

```
#------------------------------------------------------------------------------
# driver_params
#------------------------------------------------------------------------------
driver.burnin = 0
driver.epsilon = 0.02
driver.init_from = 'fiducial'
driver.init_scatter = 0.0
driver.lbfgs_epsilon = {'Nsat_mult': 0.01, 'f1h_cBs': 0.01}
driver.lbfgs_options = {'ftol': 1e-10, 'xtol': 1e-10, 'gtol': 1e-05}
driver.lbfgs_use_priors = True
driver.solver_type = 'nlopt'
driver.start_from = None
driver.test_convergence = False
#------------------------------------------------------------------------------
```

These parameters allow the user to specify which type of data is being fit, either "galaxy" or "quasar" and to pass options to the solver being used, either the `emcee` MCMC solver or the NLOPT solver. We will detail the MCMC solver (*MCMC*) and the LBFGS solver (*Nonlinear Optimization*) in the next sections.

1.12.2 MCMC

The pyRSD package wraps the `emcee` package to provide functionality for running MCMC chains to sample the posterior distributions of the model parameters.

Command-line Options

MCMC chains are run by passing the `mcmc` sub-command to the `rsdfit` executable. The calling sequence for the `mcmc` command is

```
$ rsdfit mcmc -h
usage: rsdfit [-h] [--version] {mcmc,nlopt,restart,analyze} ...

From more help on each of the subcommands, type:
rsdfit mcmc -h
rsdfit nlopt -h
rsdfit restart -h
rsdfit analyze -h mcmc
       [-h] [-m MODEL] [-p PARAMS] [--silent] -w WALKERS -i ITERATIONS
       [-n NCHAINS] -o FOLDER [--debug] [--no-save-model]
```

(continues on next page)

(continued from previous page)

```
optional arguments:
  -h, --help            show this help message and exit
  -m MODEL, --model MODEL
                        file name holding the model path
  -p PARAMS, --params PARAMS
                        file name holding the driver, theory, and data
                        parameters
  --silent              silence the standard output to the console
  -w WALKERS            number of emcee walkers to run the MCMC chain
                        (required)
  -i ITERATIONS         number of steps to run in the MCMC chain (required)
  -n NCHAINS, --nchains NCHAINS
                        number of MCMC chains to run concurrently
  -o FOLDER, --output FOLDER
                        the folder where the results will be written
                        (required)
  --debug               whether to print more info about the mpi4py.Pool
                        object
  --no-save-model       do not save the model instance
```

The main options are the parameter file, passed by the `-p` option, the directory to save results, passed by the `-o` option, and the name of a model to load, passed by the `-m` file. In addition, there are some MCMC-specific options:

1. **-w, —walkers**

   The number of `emcee` walkers to use. These walkers are responsible for exploring the relevant parameter space simulataneously.

   Note: The number of walkers must be at least twice the number of model parameters, and generally, the more walkers used the better. However, a trade-off exists since more walkers can become computationally infeasible when model evaluation is slow. In the case of the default model with 13 parameters, we recommend using 30-40 walkers.

   ---

2. **-i, —iterations**

   The number of iterations to run in the MCMC chain.

   Note: Generally, the models in the pyRSD package require >1000 iterations to achieve convergence, depending on how the MCMC sampler is initialized.

   ---

3. **–nchains**

   If the `rsdfit` is executed in parallel with multiple processes using MPI, it is possible to run multiple chains with MCMC concurrently with this option. This is ideal for testing the convergence of the sampler, as the chains will be compared statistically to determine if they have converged to a similar point in parameter space.

Initializing the MCMC Chains

The method used to initialize either the MCMC chains can be configured using the `driver.init_from` parameter. The allowed values of this parameter when using the MCMC solver are:

1. **fiducial** :

Initialize the parameters in a small ball around the fiducial parameter values, specified in the parameter file via the `fiducial` keyword for each free parameter

2. **prior** :

   Initialize the free parameters by drawing a value from the prior probability specified for each parameter, which will be either a uniform or normal distribution

3. **result** :

   Initialize the free parameters in a small ball around the best-fit parameters loaded from a previous result. In this case, the `driver.start_from` parameter should give the name of a `.npz`, which can be loaded into either a *pyRSD.rsdfit.results.LBFGSResults* or *pyRSD.rsdfit.results.EmceeResults* object.

Testing for Convergence

The convergence of the MCMC chain being run will be tested if the `driver.test_convergence` parameter is set to `True`. This test is performed using the Gelman and Rubin diagnostic, and the tolerance level for the test is specified via the `driver.epsilon` parameter. The convergence for MCMC chains is performed as

1. Remove the first half of the current chains
2. Calculate the within chain and between chain variances
3. Estimate the variance from the within chain and between chain variance
4. Calculate the potential scale reduction parameter and compare to `epsilon`

Note: Convergence testing is disabled for MCMC chains by default, as it is often easier to run a pre-defined number of iterations (passed by the `-i` flag), which will be specific to the user's problem at hand.

Recommended Practices

The `emcee` package recommends initializing its MCMC sampler in a small ball around what the user believes to be the area in parameter space close to the maximum probability. As such, we recommend either of the following initialization methods for best results:

1. Run the NLOPT solver, starting from a fiducial set of values, and then inititialize the MCMC solver with the result of that NLOPT run.

2. Initialize the MCMC solver with values drawn from the parameters' prior distribution and run a set of burn-in iterations (typically ~1000 or so). Then, initialize a second MCMC chain from the best-fit result of that burn-in period.

For more recommended practices regarding the `emcee` package, please see the FAQ on the emcee documentation here.

Finally, it is also useful for convergence reasons to run multiple chains at once in parallel. Often running two chains, independently initialized, with half the number of iterations is useful to asses if the chains have truly found the best-fit parameters.

1.12.3 Nonlinear Optimization

The pyRSD package includes a LBFGS solver to find the maximum a posteriori probability (MAP) estimates of the best-fit theory parameters.

Command-line Options

The MAP parameter values can be found by passing the `nlopt` sub-command to the `rsdfit` executable. The calling sequence for the `nlopt` command is

```
$ rsdfit nlopt -h
usage: rsdfit [-h] [--version] {mcmc,nlopt,restart,analyze} ...

From more help on each of the subcommands, type:
rsdfit mcmc -h
rsdfit nlopt -h
rsdfit restart -h
rsdfit analyze -h nlopt
       [-h] [-m MODEL] [-p PARAMS] [--silent] -i ITERATIONS -o FOLDER
       [--debug] [--no-save-model]

optional arguments:
  -h, --help            show this help message and exit
  -m MODEL, --model MODEL
                        file name holding the model path
  -p PARAMS, --params PARAMS
                        file name holding the driver, theory, and data
                        parameters
  --silent              silence the standard output to the console
  -i ITERATIONS         the maximum number of iterations to run (required)
  -o FOLDER, --output FOLDER
                        the folder where the results will be written
                        (required)
  --debug               whether to print more info about the mpi4py.Pool
                        object
  --no-save-model       do not save the model instance
```

The main options are the parameter file, passed by the `-p` option, the directory to save results, passed by the `-o` option, and the name of a model to load, passed by the `-m` file. In addition, the are maximum number of optimaztion steps should be passed via **-i, —iterations** flag.

Initializing the NLOPT Solver

The method used to initialize either the MCMC chains can be configured using the `driver.init_from` parameter. The allowed values of this parameter when using the MCMC solver are:

1. **fiducial** :

   Initialize the parameters to their fiducial values, specified in the parameter file via the `fiducial` keyword for each free parameter

2. **result** :

   Initialize the free parameters the values of the parameters from from a previous result. In this case, the `driver.start_from` parameter should give the name of a `.npz`, which can be loaded into either a *pyRSD.rsdfit.results.LBFGSResults* or *pyRSD.rsdfit.results.EmceeResults* object.

Additionally, the `driver.init_scatter` can be set to add random scatter drawn from a normal distributition with mean zero and standard deviation set by the value of `driver.init_scatter`. Specifically, this parameter gives the percent scatter to add, relative to the value of the parameter's fiducial value.

Parameter Derivatives

The LBFGS algorithm requires the derivatives of the model with respective to the free parameters, and analytic derivatives of nearly all of the parameters in the default parametrization of the *GalaxySpectrum* model are builtin into the package. However for some parameters, it is necessary to estimate their derivatives numerically. To do so, the NLOPT solver uses a central-difference numerical derivative with the step-size set by the user, depending on the value of the `driver.lbfgs_epsilon` parameter.

The `driver.lbfgs_epsilon` parameter can be specified as a single float, in which case this step size will be used for all parameters that require numerical derivatives. Alternatively, the parameter can be specified as a dictionary, providing different values of the step size for different parameters. This is particularly useful for parameters that have drastically different magnitudes in order to avoid numerical instabilities.

The Stopping Criteria

The LBFGS algorithm will stop when either the number of iterations has been reached, or if `driver.test_convergence` is set to `True`, when any of the convergence criteria are satisfied. These criteria can be specified by the user by adjusting the `driver.lbfgs_options` parameter. This parameter is a dictionary of options with the following keys:

1. **factr** :

   Stopping criterion based on the value of the objective function, given by `(f^k - f^{k+1})/max{|f^k|,|f^{k+1}|,1} <= factr * eps` where `eps` is the machine precision. Typical values for `factr` are: 1e12 for low accuracy; 1e7 for moderate accuracy; 10.0 for extremely high accuracy. Default is 1e5.

2. **gtol** :

   Stopping criterion based on the absolute value of the gradient norm of the objective function, given by `max(abs(G_k)) <= gtol`. Default is 1e-5.

Recommended Practices

For the default models and parametrization, the LBFGS algorithm typically needs ~200 or so iterations to converge to the best-fit parameters. We recommend at least this number of iterations in order to avoid converging to a local maximum of the likelihood function. If the user is worried about local optima, multiple NLOPT runs can be executed, initializing from different fiducial values each time.

The user can also test the sensitivity of the best-fit parameter values to the assumed prior distributions by setting the `driver.lbfgs_use_priors` to `False`. This will run a maximum likelihood parameter estimation, ignoring the prior probabilities. In nearly all cases, the MAP and maximum likelihood results should agree, unless for some reason the prior distribution is the dominant constraint on a given parameter.

1.12.4 API

The *pyRSD.rsdfit.FittingDriver* class is responsible for running parameter fits. It combines data and theory to run a Bayesian likelihood analysis.

The properties that describe the data and theory configuration are:

Nb	The number of data points
Np	The number of free parameters
dof	The number of degrees of freedom

Continued on next page

Table 13 – continued from previous page

model	The *model* object which returns the P(k,mu) or multipoles theory
results	The *results* object storing the fitting results

The functions that evaluate the likelihood, its derivatives, and associated statistics are:

lnprob([theta])	Set the theory free parameters, update the model, and return the log of the posterior probability function
lnlike([theta])	The log of the likelihood, equal to -0.5 * *chi2()*
minus_lnlike([theta, use_priors])	Return the negative log-likelihood, optionally including priors
grad_minus_lnlike([theta, epsilon, pool, ...])	Return the vector of the gradient of the negative log likelihood, with respect to the free parameters
chi2([theta])	The chi-squared for the specified model function
reduced_chi2()	The reduced chi squared value, using the current values of the free parameters
fisher([theta, epsilon, pool, use_priors, ...])	Return the Fisher information matrix
marginalized_errors([params, theta, fixed])	Return the marginalized errors on the specified parameters, as computed from the Fisher matrix
run(solver_type[, pool, chains_comm])	Run the whole fitting analysis, from start to finish

The user can initialize a `FittingDriver` object from a results directory using

from_directory(dirname[, results_file, ...])	Load a `FittingDriver` from a results directory

The best-fit parameters can be set and visualized using

set_fit_results([method])	Set the free parameters from the results objects and update the model
plot([usetex, ax, colors, use_labels])	Plot the best-fit theory and data points
plot_residuals()	Plot the residuals of the best-fit theory and data

class pyRSD.rsdfit.**FittingDriver**(*param_file*, *init_model=True*, ***kwargs*)

A driver to run the parameter fitting pipeline, merging together a model, theory, and fitting algorithm

Parameters **param_file** : str

a string specifying the name of the main parameter file

init_model : bool, optional

if *True*, initialize the theoretical model upon initialization; default is *True*

Nb

The number of data points

Np

The number of free parameters

chi2(*theta=None*)

The chi-squared for the specified model function

This returns

$$\chi^2 = (\mathcal{M} - \mathcal{D})^T C^{-1} (\mathcal{M} - \mathcal{D})$$

Parameters **theta** : array_like, optional

an array of the free parameters to evaluate the statistic at; if `None`, the current values of the free parameters in `theory.fit_params` is used

dof

The number of degrees of freedom

This is equal to the number of data points minus the number of free parameters

fisher(*theta=None*, *epsilon=0.0001*, *pool=None*, *use_priors=True*, *numerical=False*, *numerical_from_lnlike=False*)

Return the Fisher information matrix

This is defined as the negative Hessian of the log likelihood with respect to the parameter vector

$$F_{ij} = -\frac{\partial^2 \log \mathcal{L}}{\partial\theta_i \partial\theta_j}$$

This uses a central-difference finite-difference approximation to compute the numerical derivatives

Parameters **theta** : array_like, optional

if provided, the values of the free parameters to compute the gradient at; if not provided, the current values of free parameters from *theory.fit_params* will be used

epsilon : float or array_like, optional

the step-size to use in the finite-difference derivative calculation; default is *1e-4* – can be different for each parameter

pool : MPIPool, optional

a MPI Pool object to distribute the calculations of derivatives to multiple processes in parallel

use_priors : bool, optional

whether to include the log priors in the objective function when minimizing the negative log probability

numerical : bool, optional

if *True*, evaluate gradients of P(k,mu) numerically using finite difference

numerical_from_lnlike : bool, optional

if *True*, evaluate the gradient by taking the numerical derivative of *minus_lnlike()*

classmethod **from_directory**(*dirname*, *results_file=None*, *model_file=None*, *init_model=True*, ***kwargs*)

Load a *FittingDriver* from a results directory

This reads `params.dat` file, optionally loading a pickled model and a results object from file

Parameters **dirname** : str

the name of the directory holding the results

results_file : str, optional

the name of the file holding the results. Default is `None`

model_file : str, optional

the name of the file holding the model to load. Default is `None`

init_model : bool, optional

whether to initialize the RSD model upon loading. If a model file exists in the specified directory, the model is loaded and no new model is initialized

grad_minus_lnlike(*theta=None*, *epsilon=0.0001*, *pool=None*, *use_priors=True*, *numerical=False*, *numerical_from_lnlike=False*)

Return the vector of the gradient of the negative log likelihood, with respect to the free parameters

This uses a central-difference finite-difference approximation to compute the numerical derivatives

Parameters **theta** : array_like, optional

if provided, the values of the free parameters to compute the gradient at; if not provided, the current values of free parameters from `theory.fit_params` will be used

epsilon : float or array_like, optional

the step-size to use in the finite-difference derivative calculation; default is *1e-4* – can be different for each parameter

pool : MPIPool, optional

a MPI Pool object to distribute the calculations of derivatives to multiple processes in parallel

use_priors : bool, optional

whether to include the log priors in the objective function when minimizing the negative log probability

numerical : bool, optional

if *True*, evaluate gradients of P(k,mu) numerically using finite difference

numerical_from_lnlike : bool, optional

if *True*, evaluate the gradient by taking the numerical derivative of *minus_lnlike()*

lnlike(*theta=None*)

The log of the likelihood, equal to -0.5 * *chi2()*

Parameters **theta** : array_like, optional

an array of the free parameters to evaluate the statistic at; if `None`, the current values of the free parameters in `theory.fit_params` is used

lnprob(*theta=None*)

Set the theory free parameters, update the model, and return the log of the posterior probability function

This returns -0.5 *chi2()* + `lnprior()`

Parameters **theta** : array_like, optional

an array of the free parameters to evaluate the statistic at; if `None`, the current values of the free parameters in `theory.fit_params` is used

marginalized_errors(*params=None*, *theta=None*, *fixed=[]*, ***kws*)

Return the marginalized errors on the specified parameters, as computed from the Fisher matrix

This is given by: $(\mathcal{F}^{-1}))^(1/2)$

Optionally, we can fix certain parameters, specified in `fixed`

Parameters **params** : list, optional

return errors for this parameters; default is all free parameters

theta : array_like, optional

the array of free parameters to evaluate the best-fit model at

fixed : list, optional

list of free parameters to fix when evaluating marginalized errors

****kws :**

additional keywords passed to *fisher()*

Returns errors : array_like

the marginalized errors as computed from the inverse of the Fisher matrix

minus_lnlike(*theta=None*, *use_priors=False*)
Return the negative log-likelihood, optionally including priors

Parameters theta : array_like, optional

an array of the free parameters to evaluate the statistic at; if `None`, the current values of the free parameters in `theory.fit_params` is used

use_priors : bool, optional

whether to include the log priors in the objective function when minimizing the negative log probability

model
The *model* object which returns the P(k,mu) or multipoles theory

plot(*usetex=False*, *ax=None*, *colors=None*, *use_labels=True*, ***kws*)
Plot the best-fit theory and data points

plot_residuals()
Plot the residuals of the best-fit theory and data

reduced_chi2()
The reduced chi squared value, using the current values of the free parameters

results
The *results* object storing the fitting results

run(*solver_type*, *pool=None*, *chains_comm=None*)
Run the whole fitting analysis, from start to finish

Parameters solver_type : {'mcmc', 'nlopt'}

either run a MCMC fit with emcee or a nonlinear optimization using LBFGS

pool : MPIPool, optional

a MPI pool object to distribute tasks too

chains_comm : MPI communicator, optional

a communicator for communicating between multiple MCMC chains

set_fit_results(*method='median'*)
Set the free parameters from the results objects and update the model

1.13 Exploring the Results

Here, we describe how pyRSD saves the results of the parameter estimation step and how the user can use these results to explore the best-fitting theory and parameters.

1.13.1 MCMC

The main result of running `rsdfit mcmc ...` is a `*.npz` file saved to the output directory for each MCMC chain that was run. These results can be loaded from file using the *pyRSD.rsdfit.results.EmceeResults* class.

The *EmceeResults* stores the entire MCMC chain for each free parameter and can return statistics based on this chain for each parameter, i.e., the median, 1σ and 2σ errors, etc. This object also computes the corresponding MCMC chain for the constrained parameter values and can return information for each constrained parameter.

The median parameter values, as well as the 68% and 95% confidence intervals can be quickly displayed by printing the *EmceeResults* object. For example,

```
In [1]: from pyRSD.rsdfit.results import EmceeResults

In [2]: results = EmceeResults.from_npz('mcmc_result.npz')

# print out a summary of the parameters, with mean values and 68% and 95% intervals
In [3]: print(results)
Free parameters [ median (+/-68%) (+/-95%) ]
_________________
<Parameter Nsat_mult:      2.423 (+0.2082 -0.1809) (+0.4361 -0.3432)>
<Parameter alpha_par:      1.009 (+0.00682 -0.007681) (+0.01353 -0.01379)>
<Parameter alpha_perp:     1.005 (+0.004241 -0.004049) (+0.008281 -0.008406)>
<Parameter b1_cA:          1.999 (+0.05749 -0.05801) (+0.1126 -0.1265)>
<Parameter f:              0.867 (+0.02897 -0.02502) (+0.05642 -0.05073)>
<Parameter f1h_sBsB:       3.569 (+0.5521 -0.6841) (+1.224 -1.396)>
<Parameter fs:             0.1434 (+0.007553 -0.008046) (+0.01554 -0.01544)>
<Parameter fsB:            0.4662 (+0.08146 -0.0792) (+0.1685 -0.1606)>
<Parameter gamma_b1sA:     1.31 (+0.1001 -0.1032) (+0.2119 -0.2205)>
<Parameter gamma_b1sB:     2.343 (+0.1661 -0.181) (+0.3225 -0.3636)>
<Parameter sigma8_z:       0.5411 (+0.01224 -0.01362) (+0.02611 -0.02452)>
<Parameter sigma_c:        0.9297 (+0.06205 -0.06473) (+0.1126 -0.1177)>
<Parameter sigma_sA:       3.443 (+0.2779 -0.2702) (+0.5436 -0.5378)>

Constrained parameters [ median (+/-68%) (+/-95%) ]
_________________________
<Parameter NsBsB:          5.178e+04 (+1.721e+04 -1.36e+04) (+4.16e+04 -2.44e+04)>
<Parameter b1_sA:          2.617 (+0.169 -0.1761) (+0.3037 -0.3725)>
<Parameter sigma_sB:       5.71 (+0.3067 -0.2659) (+0.6624 -0.4928)>
<Parameter NcBs:           2.261e+04 (+2900 -2096) (+6130 -4001)>
<Parameter b1_sB:          4.686 (+0.2679 -0.3235) (+0.5596 -0.672)>
<Parameter b1_cB:          3.175 (+0.1504 -0.1736) (+0.2942 -0.3522)>
<Parameter fcB:            0.122 (+0.01336 -0.0149) (+0.02856 -0.02761)>
<Parameter b1_c:           2.141 (+0.04721 -0.04684) (+0.08834 -0.08337)>
<Parameter b1:             2.344 (+0.05369 -0.04764) (+0.1051 -0.1053)>
<Parameter fsigma8:        0.4689 (+0.009673 -0.009076) (+0.01866 -0.01848)>
<Parameter b1_s:           3.575 (+0.1907 -0.2037) (+0.3854 -0.4186)>
<Parameter alpha:          1.006 (+0.004107 -0.004331) (+0.007706 -0.0083)>
<Parameter epsilon:        0.00125 (+0.002298 -0.002462) (+0.004696 -0.004652)>
<Parameter b1sigma8:       1.268 (+0.007575 -0.008006) (+0.01531 -0.01495)>
<Parameter F_AP:           1.004 (+0.006927 -0.007386) (+0.01419 -0.01393)>
```

The EmceeParameter Object

The *EmceeResults* object provides access to the parameters via a dictionary-like behavior. When accessing parameters using the name as the key, a *EmceeParameter* object is returned. This object has several useful attributes holding information about the parameter:

1. **flat_trace** : the flattened MCMC chain for this parameter
2. **median** : the median of the trace
3. **mean**: the average value of the trace
4. **one_sigma** : tuple of the lower and upper 1σ errors
5. **two_sigma** : tuple of the lower and upper 2σ errors
6. **three_sigma** : tuple of the lower and upper 3σ errors
7. **stderr** : the average of the upper and lower 1σ error

For example,

```
In [4]: f = results['f']

In [5]: print(f.median, f.one_sigma, f.two_sigma)
0.8669514683620745 [-0.02501616327798617, 0.02896606398246815] [-0.05073347536029271,␣
↪0.05641800142862763]

In [6]: sigma8 = results['sigma8_z']

In [7]: print(sigma8.median, sigma8.one_sigma, sigma8.two_sigma)
0.5410678224052163 [-0.013623734433715451, 0.01224434253786777] [-0.02452235949056536,
↪ 0.026109037129819712]

# access to constrained parameters too
In [8]: fs8 = results['fsigma8']

In [9]: print(fs8)
<Parameter fsigma8:        0.4689 (+0.009673 -0.009076) (+0.01866 -0.01848)>
```

Specifying the Burn-in Period

The user can specify a "burn-in" period for a given results object, by changing the `burnin` attribute of the *`EmceeResults`* object. The value of this attribute specifies the number of steps to ignore, starting from the beginning of the MCMC chain. The ignored steps are not included when computing any statistics from the parameter traces.

The attributes of the *`EmceeParameter`* object automatically take into account the value of the burnin attribute. Thus, the user just needs to set the `burnin` and the parameter values will automatically adjust accordingly.

For example,

```
In [10]: results.burnin = 0 # ignore 0 steps

In [11]: print(results['fsigma8'].median)
0.4689141809940338

In [12]: results.burnin = 500 # ignore the first 500 steps

In [13]: print(results['fsigma8'].median) # slight change in value
0.468679279088974
```

Ideally, if the chain has converged for a given parameter, the user should see little change to the parameter's value when adjusting the burnin period.

The Best-fit Values

The user can quickly access the best-fit parameter vector using the `values` attribute, which returns the median value of each free parameter. Similarly, the `constrained_values` attribute returns the median value of each constrained parameter.

Alternatively, the user can access the parameter vector that maximizes the log probability by accessing the `max_lnprob_values` and `max_lnprob_constrained_values` attributes.

1.13.2 Nonlinear Optimization

The main result of running `rsdfit nlopt ...` is a `*.npz` file saved to the output directory for each MCMC chain that was run. These results can be loaded from file using the *pyRSD.rsdfit.results.LBFGSResults* class.

The *LBFGSResults* stores the best-fit values for both the free parameters and the constrained parameters, as computed from the final iteration of the LBFGS algorithm.

The best-fit parameter values and the corresponding minimum $\chi^2 value$ can be quickly displayed by printing the *LBFGSResults* object. For example,

```
In [1]: from pyRSD.rsdfit.results import LBFGSResults

In [2]: results = LBFGSResults.from_npz('nlopt_result.npz')

# print out a summary of the best-fit parameters
In [3]: print(results)
minimum chi2 = 120.57745038002743

Free parameters [ mean ]
________________
Nsat_mult       : 2.4324089012014207
alpha_par       : 1.0072991239946898
alpha_perp      : 1.0048333693698863
b1_cA           : 2.0068377074748778
f               : 0.8735346371321935
f1h_sBsB        : 3.6140366462767153
fs              : 0.14175570928456935
fsB             : 0.4782779433107577
gamma_b1sA      : 1.31213984320964
gamma_b1sB      : 2.3651886897525896
sigma8_z        : 0.5363993024676117
sigma_c         : 0.9194602113178405
sigma_sA        : 3.420304679802984

Constrained parameters [ mean ]
_____________________________
NsBsB           : 51736.1
b1_sA           : 2.6332517
NcBs            : 23183.68
fcB             : 0.11864934
sigma_sB        : 5.739127
b1_sB           : 4.74655
b1_cB           : 3.2117057
epsilon         : 0.0008172965
b1_c            : 2.1497946
b1sigma8        : 1.2667638
```

(continues on next page)

(continued from previous page)

```
alpha           : 1.0056546
F_AP            : 1.0024539
b1_s            : 3.6439955
fsigma8         : 0.46856338
b1              : 2.3616061
```

The minimum χ^2 value can be accessed from the `min_chi2` attribute of the *LBFGSResults* object.

Parameter Access

The *LBFGSResults* object provides access to the parameters via a dictionary-like behavior. When accessing parameters using the name as the key, the best-fit value of that parameter is returned.

For example,

```
# growth rate
In [4]: f = results['f']

In [5]: print(f)
0.8735346371321935

# power spectrum normalization
In [6]: sigma8 = results['sigma8_z']

In [7]: print(sigma8)
0.5363993024676117

# this is the product of f and sigma8_z
In [8]: fs8 = results['fsigma8']

In [9]: print(fs8)
0.46856338

In [10]: print(numpy.isclose(fs8, f*sigma8))
\\\\\\\\\\True
```

The Best-fit Values

The user can quickly access the best-fit parameter vector using the `min_chi2_values` attribute, which returns the value of each free parameter at the minimum χ^2 of the fit. Similarly, the `min_chi2_constrained_values` attribute returns the corresponding value of each constrained parameter at the minimum χ^2.

1.13.3 Visualizing the Results

Comparing the Best-fit Model to Data

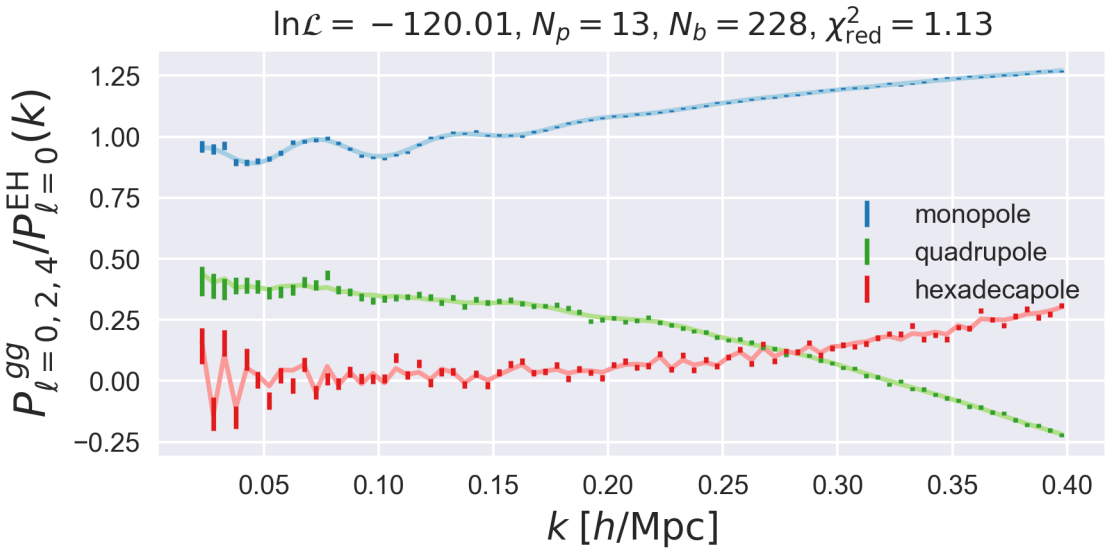
The best-fitting theory can be compared to the data visually by loading the results of a fitting run using the *pyRSD.rsdfit.FittingDriver* and using the *FittingDriver.plot* function. This function will plot the data and best-fitting power spectra, properly normalized by a linear power spectrum. For example,

```
from pyRSD.rsdfit import FittingDriver
```

(continues on next page)

(continued from previous page)

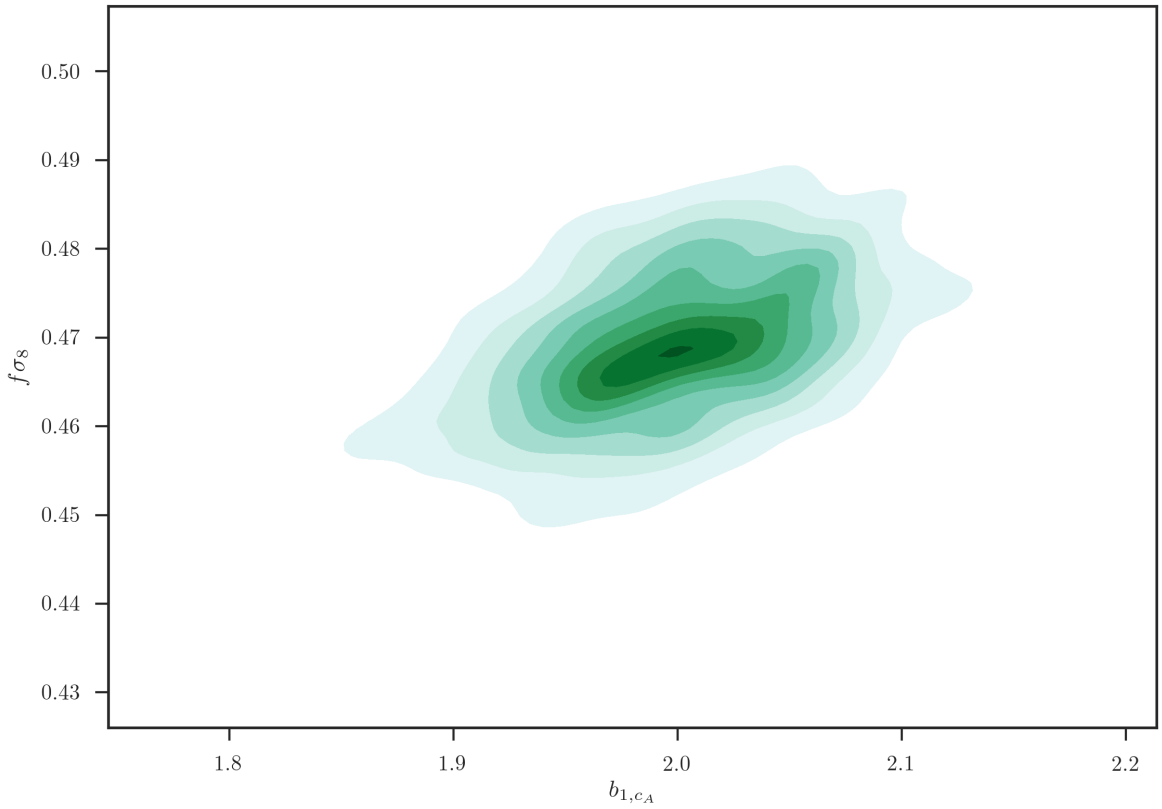
```
# load the model and results into one object
d = FittingDriver.from_directory('pyRSD-example', model_file='pyRSD-example/model.npy
↪', results_file='pyRSD-example/nlopt_result.npz')

# set the fit results
d.set_fit_results()

# make a plot of the data vs the theory
d.plot()
show()
```



Visualizng MCMC Chains

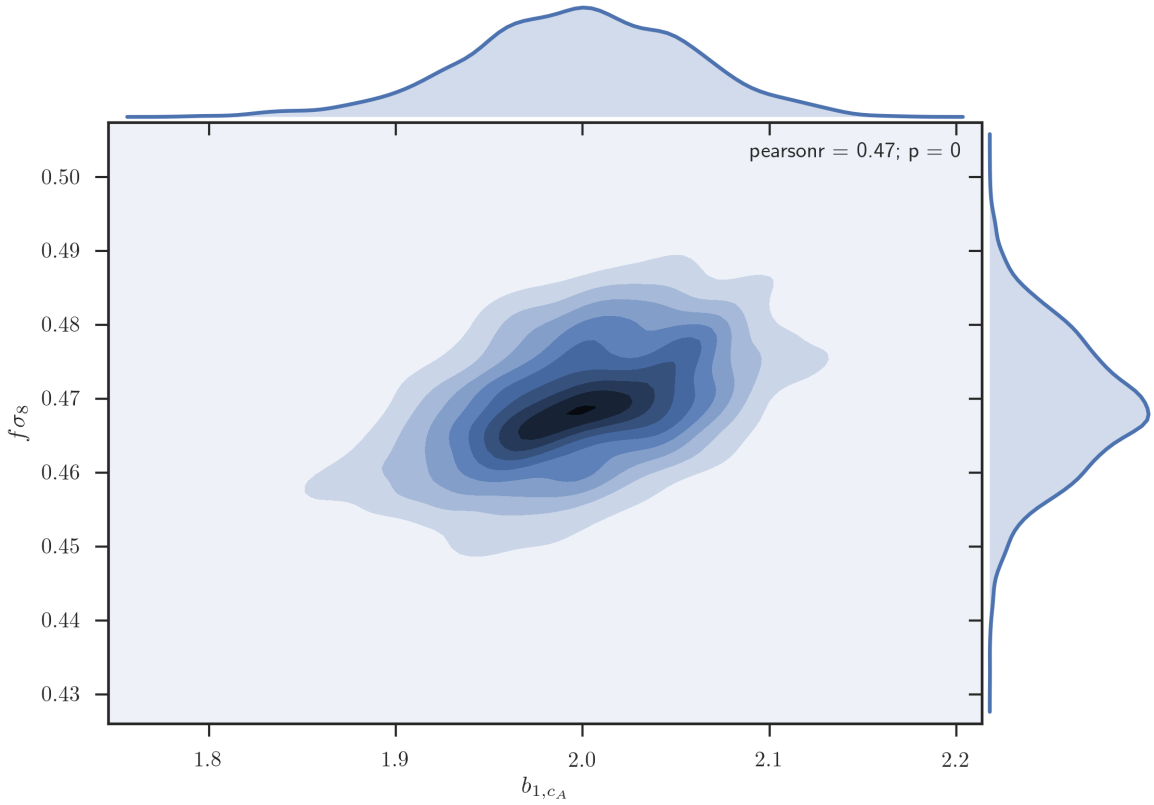
The user can plot 2D correlations between parameters using the *EmceeResults.kdeplot_2d()* function, which uses the `seaborn.kdeplot()` function,

```
from pyRSD.rsdfit.results import EmceeResults
r = EmceeResults.from_npz('mcmc_result.npz')

# 2D kernel density plot
r.kdeplot_2d('b1_cA', 'fsigma8', thin=10)
```



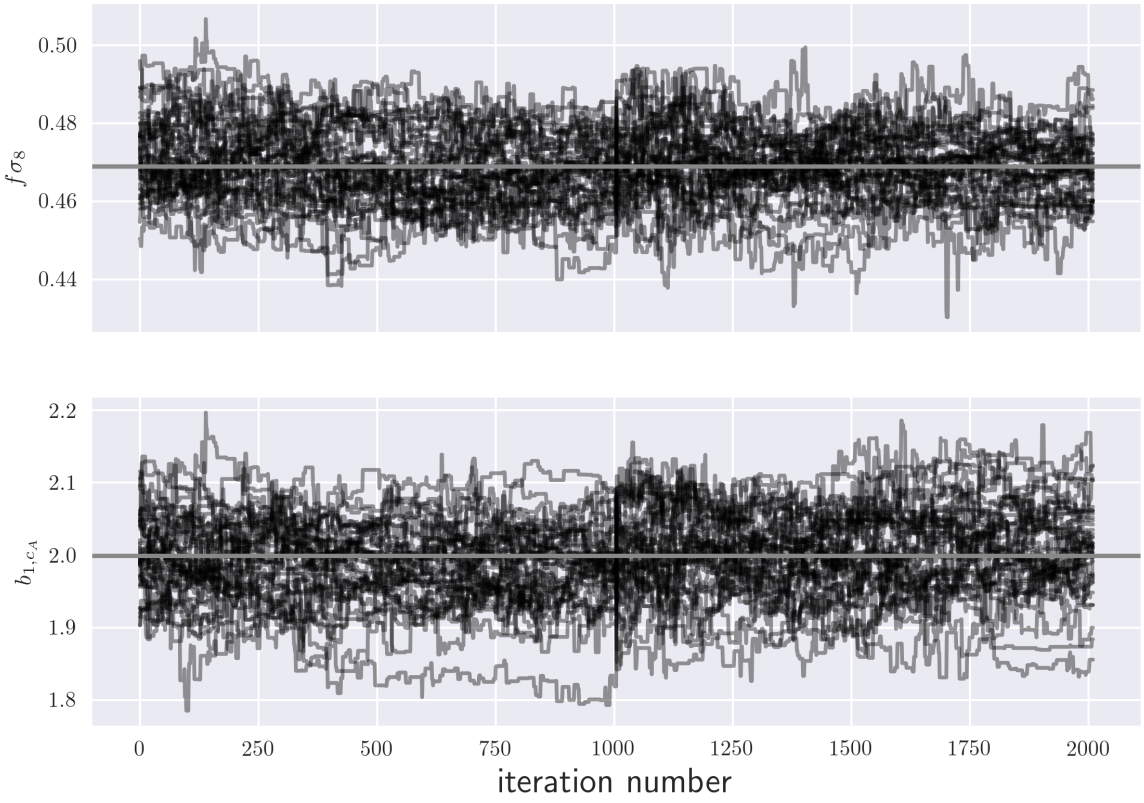
A joint 2D plot of the MCMC chains with 1D histograms can be plotted using the *EmceeResults.jointplot_2d()*, which uses the `seaborn.jointplot()` function

```
# 2D joint plot
r.jointplot_2d('b1_cA', 'fsigma8', thin=10)
```



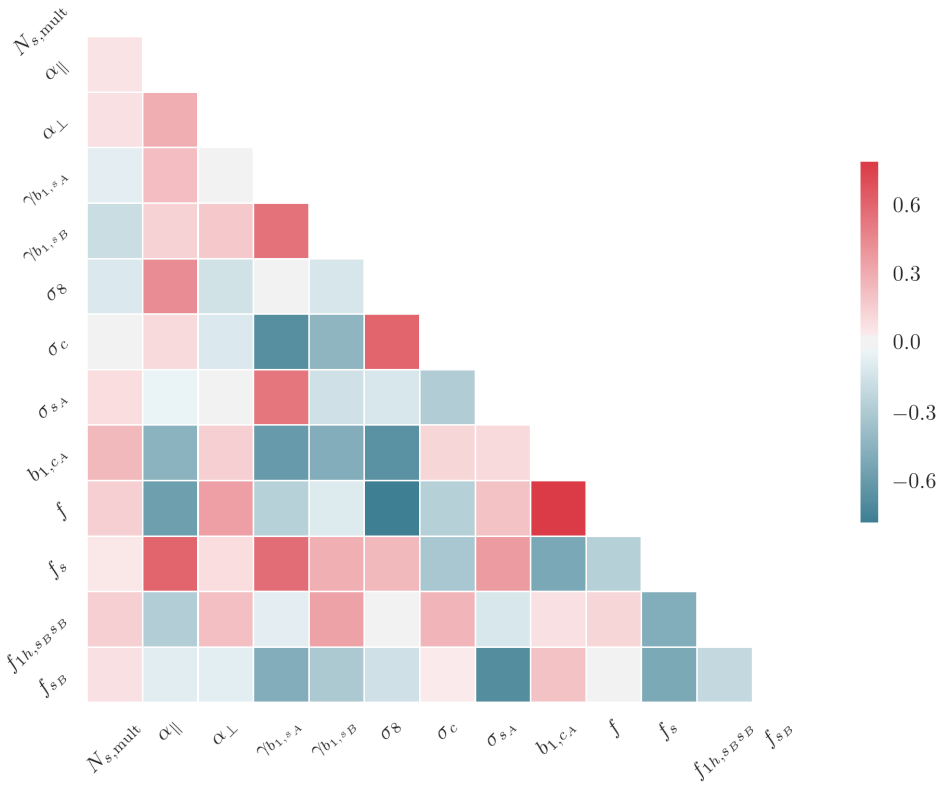
In order to investigate whether the chains have converged, the user can plot the timeline of the MCMC chain for a given parameter using the *EmceeResults.plot_timeline()* function

```
# timeline plot
r.plot_timeline('fsigma8', 'b1_cA', thin=10)
```



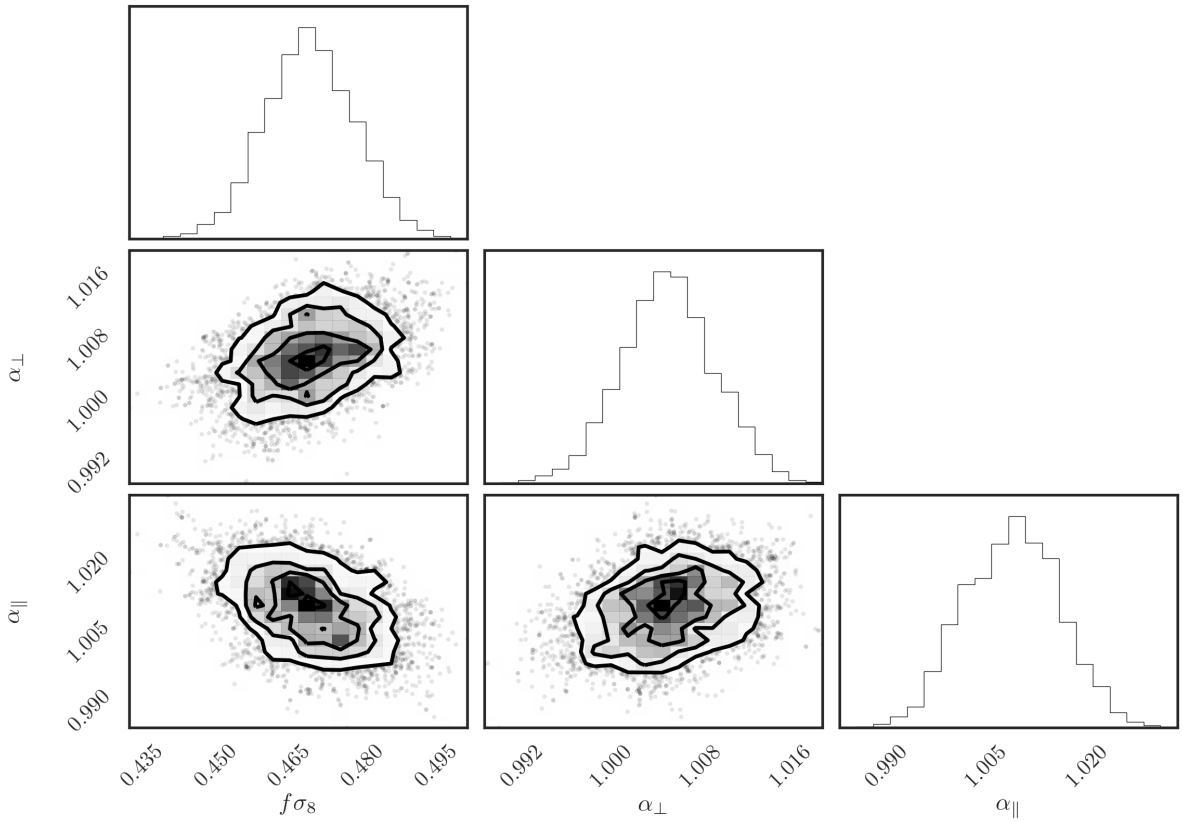
The correlation matrix between parameters can be plotted using the *EmceeResults.plot_correlation()* function, which uses the seaborn.heatmap() function

```
# correlation between free parameters
r.plot_correlation(params='free')
```



And, finally, a triangle plot of 2D and 1D histograms for the desired parameters can be produced using the *EmceeResults.plot_triangle()*, which relies on the corner.corner() function

```
# make a triangle plot
r.plot_triangle('fsigma8', 'alpha_perp', 'alpha_par', thin=10)
```



1.13.4 API

MCMC Results

The best-fit parameter vector can be accessed from the MCMC chain using the following functions of the *EmceeResults* object:

EmceeResults.values()	Convenience function to return the median values for the free parameters
EmceeResults.constrained_values()	Convenience function to return the median values for the constrained
EmceeResults.max_lnprob_values()	Return the value of the free parameters at the iteration with the maximum
EmceeResults.max_lnprob_constrained_values()	Return the value of the constrained parameters at the iteration
EmceeResults.peak_values()	Convenience function to return the peak values for the free parameters
EmceeResults.peak_constrained_values()	Convenience function to return the peak values for the

The results object can be saved and loaded from file using:

EmceeResults.to_npz(filename, **meta)	Save the relevant information of the class to a numpy `npz` file
EmceeResults.from_npz(filename)	Load a numpy `npz` file and return the corresponding `EmceeResults` object

Correlations between parameters can be analyzed using:

EmceeResults.sorted_1d_corrs([params, ...])	Return a pandas DataFrame holding the correlation matrix,
EmceeResults.corr([params, labels, use_latex])	Return a pandas DataFrame holding the correlation matrix,

The results can be visualized with the following function, which take advantage of the `seaborn` Python plotting module:

EmceeResults.jointplot_2d(param1, param2[, ...])	Plot the 2D traces of the given parameters, using KDE via the `seaborn.jointplot()` function.
EmceeResults.kdeplot_2d(param1, param2[, ...])	Plot the 2D traces of the given parameters, using KDE via `seaborn.kdeplot()`
EmceeResults.plot_correlation([params, ...])	Plot the diagonal correlation matrix, using `seaborn.heatmap`
EmceeResults.plot_timeline(*names, **kwargs)	Plot the chain timeline for as many parameters as specified in the *names* tuple.
EmceeResults.plot_triangle(*names, **kwargs)	Make a triange plot for the desired parameters using the `corner.corner()` function.

class pyRSD.rsdfit.results.**EmceeResults**(*sampler*, *fit_params*, *burnin=None*, ***meta*)

Class to hold the fitting results from an *emcee* MCMC run

as_dict(*kind=None*)

Return a dictionary of the values, either the *median*, *peak*, or *max_lnprob*

burnin

The number of iterations to treat as part of the "burnin" period, where the chain hasn't stabilized yet

chains(*params=[]*, *labels={}*, *use_latex=False*)

Return a pandas DataFrame holding the chains (flat traces) as columns for both the free and constrained parameters

Note that any burnin is removed from the chains.

Parameters **params** : list, {'free', 'constrained'}, optional

return only a subset of the parameters

use_latex : bool, optional

If *True*, try to use any available latex names for the tick labels; default is *False*

Returns DataFrame :

the DataFrame holding the flat traces for all parameters

constrained_values()

Convenience function to return the median values for the constrained parameters as an array

copy()

Return a deep copy of the *EmceeResults* object

corr (*params=[]*, *labels={}*, *use_latex=False*)
Return a pandas DataFrame holding the correlation matrix, as computed from the `chains` DataFrame, for all parameters

Parameters **params** : list, {'free', 'constrained'}, optional

return only a subset of the parameters

use_latex : bool, optional

If *True*, try to use any available latex names for the tick labels; default is *False*

Returns DataFrame (Np, Np) :

the DataFrame holding the correlation between parameters

error_rescaling
Rescale error on parameters due to covariance matrix from mocks

classmethod from_npz (*filename*)
Load a numpy `npz` file and return the corresponding `EmceeResults` object

iterations
The number of iterations performed, as computed from the chain

jointplot_2d (*param1*, *param2*, *thin=1*, *rename={}*, *crosshairs={}*, ***kwargs*)
Plot the 2D traces of the given parameters, using KDE via the `seaborn.jointplot()` function.

Parameters **param1** : str

the name of the first parameter

param2 : str

the name of the second parameter

thin : int, optional

thin the plotted array randomly by this amount

rename : dict, optional

dictionary giving a string to rename each variable; default will try to use any latex names stored

crosshairs : dict, optional

values to show as horizontal or vertical lines

****kwargs :**

additional keyword arguments to pass to `seaborn`

Notes

Any iterations during the "burnin" period are excluded

kdeplot_2d (*param1*, *param2*, *thin=1*, *rename={}*, *crosshairs={}*, ***kwargs*)
Plot the 2D traces of the given parameters, using KDE via `seaborn.kdeplot()`

Parameters **param1** : str

the name of the first parameter

param2 : str

the name of the second parameter

thin : int, optional

thin the plotted array randomly by this amount

rename : dict, optional

dictionary giving a string to rename each variable; default will try to use any latex names stored

crosshairs : dict, optional

values to show as horizontal or vertical lines

****kwargs :**

additional keyword arguments to pass to `seaborn`

Notes

Any iterations during the "burnin" period are excluded

max_lnprob
The value of the maximum log probability

max_lnprob_constrained_values()
Return the value of the constrained parameters at the iteration with the maximum probability

max_lnprob_values()
Return the value of the free parameters at the iteration with the maximum probability

ndim
The number of free parameters

peak_constrained_values()
Convenience function to return the peak values for the constrained parameters as an array

peak_values()
Convenience function to return the peak values for the free parameters as an array

plot_correlation(*params=[]*, *use_latex=True*, *labelsize=10*)
Plot the diagonal correlation matrix, using `seaborn.heatmap`

Parameters **params** : list, {'free', 'constrained'}, optional

return only a subset of the parameters

use_latex : bool, optional

If *True*, try to use any available latex names for the tick labels; default is *True*. You might want to set this to *False* if you are trying to use mpld3

plot_timeline(**names*, ***kwargs*)
Plot the chain timeline for as many parameters as specified in the *names* tuple.

This plots the value of each walker as a function of iteration.

Parameters **names** : tuple

The string names of the parameters to plot

outfile : str, optional

If not *None*, save the resulting figure with the specified name

Returns **fig** : matplotlib.Figure

The figure object

Notes

Any iterations during the "burnin" period are excluded

plot_triangle(**names*, ***kwargs*)
Make a triange plot for the desired parameters using the `corner.corner()` function.

Parameters **names** : tuple

The string names of the parameters to plot

thin : int, optional

The factor to thin the number of samples plotted by. Default is 1 (plot all samples)

outfile : str, optional

If not *None*, save the resulting figure with the specified name

Returns **fig** : matplotlib.Figure

The figure object

Notes

Any iterations during the "burnin" period are excluded

sorted_1d_corrs(*params=[]*, *use_latex=False*, *absval=False*)
Return a pandas DataFrame holding the correlation matrix, as computed from the `chains` DataFrame, for all parameters

Parameters **params** : list, {'free', 'constrained'}, optional

return only a subset of the parameters

use_latex : bool, optional

If *True*, try to use any available latex names for the tick labels; default is *False*

Returns DataFrame (Np, Np) :

the DataFrame holding the correlation between parameters

summarize_fit()
Summarize the fit, by plotting figures and outputing the relevant information

to_npz(*filename*, ***meta*)
Save the relevant information of the class to a numpy `npz` file

values()
Convenience function to return the median values for the free parameters as an array

verify_param_ordering(*free_params*, *constrained_params*)
Verify the ordering of *EmceeResults.chain*, making sure that the chains have the ordering specified by *free_params* and *constrained_params*

walkers
The number of walkers, as computed from the chain

class pyRSD.rsdfit.results.**EmceeParameter**(*name*, *trace*, *burnin=0*)
Class to hold the parameter fitting result

burnin
The number of iterations to exclude as part of the "burn-in" phase

error_rescaling
Rescale errors by this factor

fiducial
The fiducial value of the parameter

flat_trace
Returns the flattened chain, excluding steps that occured during the "burnin" period

mean
Return the average value of the chain

median
Return the median of the trace, i.e., the 50th percentile of the trace

one_sigma
Return the lower and upper one-sigma error intervals

These are computed from the percentiles, `50 - 15.86555` and `84.13445 - 50`

Returns lower, upper

The lower and upper 1-sigma error intervals

peak
Return the value of the parameter that gives the peak of the posterior PDF, as determined through Gaussian kernel density estimation

stderr
The one-sigma standard error, averaging the lower and upper bounds

three_sigma
Return the lower and upper three-sigma error intervals

There are computed from the percentiles, `50 - 0.135` and `99.865 - 50`

Returns lower, upper

The lower and upper 3-sigma error intervals

trace(*niter=None*)
Return the sample values at a specific iteration number.

shape: (nwalkers, niters)

two_sigma
Return the lower and upper two-sigma error intervals.

These are computed from the percentiles, `50 - 2.2775` and `97.7225 - 50`

Returns lower, upper

The lower and upper 2-sigma error intervals

NLOPT Results

class pyRSD.rsdfit.results.**LBFGSResults**(*data*, *fit_params*)
Class to hold the fitting results from an *scipy.optimize* L-BFGS-B nonlinear optimization run.

copy()
Return a deep copy of the *LBFGSResults* object

classmethod from_npz(*filename*)
Load a numpy `npz` file and return the corresponding *LBFGSResults* object

iterations
The total number of iterations ran

ndim
The number of free parameters

summarize_fit(**args*, ***kwargs*)
Summarize the fit by printing self

to_npz(*filename*)
Save the relevant information of the class to a numpy `npz` file

verify_param_ordering(*free_params*, *constrained_params*)
Verify the ordering of of parameters, making sure that the ordering specified by *free_params* and *constrained_params* is respected in the results

1.14 Advanced Patterns

Here, we describe how pyRSD saves the results of the parameter estimation step and how the user can use these results to explore the best-fitting theory and parameters.

1.14.1 Restarting Parameter Fits

The `rsdfit` executable includes a `restart` sub-command for restarting parameter fits from existing parameter fit, which can be either a MCMC or NLOPT `.npz` result file. To restart from a specific result, simply pass the name of the result file and specify the appropriate model to load and the number of additional iterations to run.

The calling sequence is:

```
$ rsdfit restart -h
usage: rsdfit [-h] [--version] {mcmc,nlopt,restart,analyze} ...

From more help on each of the subcommands, type:
rsdfit mcmc -h
rsdfit nlopt -h
rsdfit restart -h
rsdfit analyze -h restart
       [-h] -m MODEL -i ITERATIONS [-b BURNIN] [--silent] [--debug]
       restart_files [restart_files ...]

positional arguments:
  restart_files         the name of the existing results file to restart from;
                        multiple files can be restarted at once, but as many
                        parallel processes as files must be present

optional arguments:
  -h, --help            show this help message and exit
  -m MODEL, --model MODEL
                        file name holding the model path to load (required)
  -i ITERATIONS         the number of additional iterations to run (required)
```

(continues on next page)

(continued from previous page)

```
  -b BURNIN             the number of steps to consider burnin, if running
                        MCMC
  --silent              silence the standard output to the console
  --debug               whether to print more info about the mpi4py.Pool
                        object
```

The code will run the number of additional iterations specified by the user via the `-i` flag. Upon finishing, a new results file holding the concatenation of the result that was restarted and the new result will be written to file. Then, lastly, the original result file that was restarted will be deleted.

Note: It is possible to specify multiple result files on the command line to restart. In this case, a parameter fit will be restarted for each file passed, and the fits will run in parallel. Thus, there must be enough parallel MPI processes available to run each restart file specified on the command line. If this isn't the case, the code will crash with an error.

1.14.2 Analyzing MCMC Results

The `rsdfit` executable includes an `analyze` sub-command that can run some basic analysis of the MCMC chains in a given directory, including automatically generating nice plot results, removing burn-in steps, and writing out best-fit parameter files.

The calling sequence is:

```
$ rsdfit analyze -h
usage: rsdfit [-h] [--version] {mcmc,nlopt,restart,analyze} ...

From more help on each of the subcommands, type:
rsdfit mcmc -h
rsdfit nlopt -h
rsdfit restart -h
rsdfit analyze -h analyze
       [-h] [--minimal] [--bins BINS] [--show-mean] [--noplot] [--noplot-2d]
       [--show-fiducial] [--burnin BURNIN] [--contours-only] [--all]
       [--ext {pdf,png,eps}] [--fontsize FONTSIZE] [--ticksize TICKSIZE]
       [--line-width LINE_WIDTH] [--decimal DECIMAL] [--ticknumber TICKNUMBER]
       [--thin THIN] [--rescale-errors] [--extra OPTIONAL_PLOT_FILE]
       files [files ...]

positional arguments:
  files                 files to analyze: either a file(s), or a complete
                        directory name

optional arguments:
  -h, --help            show this help message and exit
  --minimal             use this flag to avoid computing the posterior
                        distribution (default: False)
  --bins BINS           number of bins in the histograms used to derive
                        posterior probabilities. Decrease this number for
                        smoother plots at the expense of masking details.
                        (default: 20)
  --show-mean           remove the mean likelihood from the 1D posterior plots
                        (default: False)
  --noplot              do not produce any plot, simply compute the posterior
                        (default: True)
```

(continues on next page)

(continued from previous page)

```
  --noplot-2d           produce only the 1d posterior plot (default: True)
  --show-fiducial       don't include fiducial lines on 1D posterior plots
                        (default: False)
  --burnin BURNIN, -b BURNIN
                        the fraction of samples to consider burnin (default:
                        None)
  --contours-only       do not fill the contours on the 2d plot (default:
                        False)
  --all                 output every subplot and data in separate files
                        (default: False)
  --ext {pdf,png,eps}   change the extension for the output file (default:
                        pdf)
  --fontsize FONTSIZE   the desired fontsize of output fonts (default: 16)
  --ticksize TICKSIZE   the tick size on the plots (default: 14)
  --line-width LINE_WIDTH
                        the linewidth of 1d plots (default: 4)
  --decimal DECIMAL     number of decimal places on ticks (default: 3)
  --ticknumber TICKNUMBER
                        number of ticks on each axis (default: 3)
  --thin THIN           the thinning factor to use (default: 1)
  --rescale-errors      whether to rescale errors (default: False)
  --extra OPTIONAL_PLOT_FILE
                        extra file to customize the output plots. You can
                        actually set all the possible options in this file,
                        including line-width, ticknumber, ticksize, etc... You
                        can specify four fields, `info.redefine` (dict with
                        keys set to the previous variable, and the value set
                        to a numerical computation that should replace this
                        variable), `info.to_change` (dict with keys set to the
                        old variable name, and value set to the new variable
                        name), `info.to_plot` (list of variables with new
                        names to plot), and `info.new_scales` (dict with keys
                        set to the new variable names, and values set to the
                        number by which it should be multiplied in the graph).
                        For instance, .. code::
                        analyze.to_plot=['name1','name2','newname3',...] analy
                        ze.new_scales={'name1':number1,'name2':number2,...}
                        (default: )
```

The user can specify a results directory as the only positional argument, or one or more *EmceeResults* file names as the positional arguments. In the case of a directory, all valid `.npz` files in that directory will be analyzed.

The steps performed by the `analyze` sub-command are:

1. The first thing the code does is compare the convergence of all parameters in the result files (both free and constrained) using the Gelman-Rubin criteria and prints out this convergence. The best results are achieved when multiple, independent, results files are provided on the command-line.

2. Next, the code removes automatically trims the chains of the burn-in steps, removing iterations that are too far away from the maximum probability. Alternatively, the user can specify the fraction of initial samples to consider burnin via the `-b, ---burnin` flag.

3. After the burn-in steps are removed, a single MCMC chain is created, and written to the `info/combined_result.npz` path. Additionally, summary files about the best-fit parameters are saved to the `info` directory.

4. Several plots are generated, based on the options specified by the user on the command line. These figures are saved to the `plots` directory. The possible plots include figures of the 1D histograms and triangle plots showing the 2D

correlations between parameters.

The user can specify groupings of parameters to plot by specifying the `analyze.to_plot_1d` and `analyze.to_plot_2d` parameters in a file and passing the name of that file via the `---extra` command line option. For example, the file may include:

```
analyze.to_plot_2d = {'biases': ['b1_cA', 'b1_cB', 'b1_sA', 'b1_sB'], \
                      'fractions' : ['fs', 'fsB', 'fcB'], \
                      'cosmo' : ['f', 'sigma8_z', 'fsigma8', 'alpha_par', 'alpha_perp
↪', 'b1sigma8', 'alpha', 'epsilon'], \
                      'sigmas' : ['sigma_c', 'sigma_s', 'sigma_sA', 'sigma_sB'], \
                      'nuisance' : ['Nsat_mult', 'f1h_sBsB', 'f1h_cBs', 'gamma_b1sB',
↪'gamma_b1sA']}
```

In this case, 2D triangle plots comparing each of these parameter groupings will be generated and saved in the `plots` directory.

Note: See the documentation of *`AnalysisDriver`* below for the accepted parameters that can be specified in the `extra` parameter file passed to the `rsdfit analyze` command.

API

class pyRSD.rsdfit.analysis.driver.**AnalysisDriver**(***kwargs*)

A class to serve as the driver for analyzing MCMC chains

__init__(***kwargs*)

Initialize the object by passing key/value pairs

Parameters **files** : list of str

list of a directory or series of files to analyze

minimal : bool, optional (*False*)

if *True*, only write the covmat and bestfit, without computing the posterior or making plots.

bins : int, optional (20)

number of bins in the histograms used to derive posterior

mean_likelihood : bool, optional (*True*)

show the mean likelihood on the 1D posterior plots

plot : bool, optional (*True*)

if *False*, do not make any plots, simply compute the posterior

plot_2d : bool, optional (*True*)

if *False*, do not produce the 2D posterior plots

contours_only : bool, optional (*False*)

if *True*, do not fill the contours on the 2d plots

subplot : bool, optional (*False*)

if *True*, output every subplot and data in separate files

extension : {'pdf', 'png', 'eps'}, optional (*pdf*)

the extension to use for output plots

fontsize : int, optional (16)

the fontsize to use on the plots

ticksize : int, optional (14)

the ticksize to use on the plots

line_width : int, optional (4)

the line-width of 1D plots

decimal : int, optional (3)

the number of decimal places on ticks

ticknumber : int, optional (3)

the number of ticks on each axis

optional_plot_file : str, optional (“”)

extra file to customize the output plots

tex_names : dict, optional, (*{}*)

dict holding a latex name to use for each parameter

to_plot_1d : list, optional (*[]*)

list of parameters to plot 1D posteriors of

to_plot_2d : dict, optional (*{}*)

dict holding groups of parameters to make 2D plots of

scales : dict, optional (*{}*)

dict holding the rescaling factors that the posterior will be divided by

save_output : bool, optional (*True*)

if *False*, do not save any output or make new directories

show_fiducial : bool, optional (*True*)

whether to show the fiducial values as vertical lines on the 1D posterior plots

fiducial : dict, optional {*{}*}

a dictionary holding fidicual values to use, which will override the original fiducial values

burnin : float, optional (*None*)

the fraction of samples to consider burnin

CHAPTER 2

Get in touch

Report bugs, suggest feature ideas, or view the source code on GitHub.

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

T

U

V

W

Z